An Interactive Graphic Programming Environment

MAX

DEVELOPMENT PACKAGE

by Miller Puckette
and David Zicarelli

Additional objects •
Cort Lippe
Chris Muir
Zack Settel
David Yadegari

Manual •
Christopher Dobrian
David Zicarelli

MAX is named for •
Max V. Mathews

OPCODE SYSTEMS, INC
3641 HAVEN DRIVE, STE. A
MENLO PARK, CA
94025-1010 USA
TEL 415-369-8131
FAX 415-369-1747

COPYRIGHT © 1990 IRCAM
AND OPCODE SYSTEMS, INC.
ALL RIGHTS RESERVED.
THIS DOCUMENT MAY
NOT, IN WHOLE OR IN PART,
BE COPIED, PHOTOCOPIED,
TRANSLATED, OR CONVERTED
TO ANY ELECTRONIC
OR MACHINE READABLE
FORM WITHOUT THE
PRIOR CONSENT OF IRCAM
AND OPCODE SYSTEMS, INC.

Contents

Introduction 4
What is Max? **4**
How To Use This Manual **5**

Basics 7
Setting Up **7**
Elements of Max **10**
Max and MIDI **21**
Complex Patches **23**

Tutorial 27
1. Saying "Hello!" **28**
2. bang means "Do It!" **32**
3. About numbers **34**
4. Using **metro** **36**
5. **toggle** and **comment** **38**
6. Test 1: Printing **40**
7. Right-to-left order **41**
8. Doing math in Max **43**
9. Using the **slider** **46**
10. Number boxes **49**
11. Test 2: Temperature conversion **53**
12. Sending and receiving MIDI notes **56**
13. Managing note data **60**
14. Sliders and dials **63**
15. Making decisions with comparisons **67**
16. More MIDI ins and outs **72**
17. Gates and switches **76**
18. Test 3: Comparisons and decisions **80**
19. Screen aesthetics **82**
20. Using the Macintosh keyboard **86**
21. Storing numbers **91**
22. Delay lines **97**
23. Test 4: Imitating a performance **101**
24. **send** and **receive** **104**
25. Managing messages **106**
26. The **patcher** object **111**
27. Your object **114**
28. Your argument **118**
29. Test 5: Probability object **122**
30. Number groups **127**
31. Using timers **132**
32. The **table** object **137**
33. Probability tables **144**
34. Managing raw MIDI data **149**
35. **seq** and **follow** **157**
36. Multi-track sequencing **164**

Contents

37. Data structures **168**
38. **expr** and **if** **179**
39. Mouse control **184**
40. Automatic actions **188**

Topics 192

Arguments **192**
Data Structures **195**
Debugging **197**
Efficiency **199**
Encapsulation **202**
Errors **204**
Externals **210**
Graphics **211**
Librarians **214**
Loops **223**
Menus **226**
MIDI **235**
Objects **238**
Ports **241**
Punctuation **243**
Quantile **245**
Sequencing **247**
Shortcuts **249**
Tables **252**

Objects 255

Index 470

Introduction

What is Max?

"Max is a graphical music programming environment for people who have hit the limits of the usual sequencer and voicing programs for MIDI equipment."

—Miller Puckette, Max reference manual, 1988

Max was developed at the *Institut de Recherche et de Coordination Acoustique/Musique* (IRCAM) in Paris, beginning in late 1986. The principal author of Max is Miller Puckette, a native of Tennessee who was trained as a mathematician and has been at the forefront of real-time computer music applications for a number of years. Originally a non-graphical language intended to control IRCAM's powerful 4X synthesizer, Max was later implemented as a graphical environment for MIDI on the Macintosh. The present version of Max has been further developed by David Zicarelli since July of 1989.

Max lets you control your equipment in any way you want. You can create applications for composing music, improvising music, providing accompaniment as you play, sending commands to synthesizers, modifying synthesizer patches—anything you can imagine doing with a computer and MIDI. Because Max turns all control information into a simple stream of numbers, you can "patch" anything to anything else.

Max provides you with a high level, graphical programming language that takes full advantage of the Macintosh's graphic capabilities and icon-based user interface. Programs are "written" using graphical objects rather than text. This reduces the need to learn a lot of arcane commands and syntax, and it provides a clear and intuitive way to write programs simply by connecting objects to each other.

Max takes care of all the low level programming tasks for you. It will trigger events at any arbitrary time in the future, interface to MIDI and other communication protocols, and perform useful logical operations.

Applications made with Max run in real time. Because of its speed, Max enables you to write programs which generate music instantly based on what you play, or which modify your performance as you play.

Max is based on the C programming language. Max provides a simple yet versatile, high level, graphical language which is itself written in C, but will be easy to use for those familiar with almost any other programming language, or even for those who have never programmed before. For those who are fluent in the C language, however, Max can be combined with C code that you write. So, if there's something you need to do that Max can't do—and Max can do a lot—you can write your own Max objects in C.

Introduction

How To Use This Manual

The Max User's Manual is comprised of four sections:

- *Basics* is an overview of all the different elements of Max.
- *Tutorial* takes you through a step-by-step course on how to program with Max.
- *Topics* contains technical information on menu commands, file management, and other aspects of Max programming.
- *Objects* contains reference information for each of the built-in and external objects supplied with Max.

Basics

Read this section first. If you have very little experience with computer programming or MIDI, you might not understand everything you read in *Basics*. Never fear. The *Tutorial* will cover topics in much greater detail, starting from square one. Max is a complex application and we don't expect everyone to "get" everything immediately.

On the other hand, if you are already knowledgeable about MIDI, have some prior programming experience, or are just an impatient over-achiever who likes to plunge into things and hates reading manuals, you should be able to get started after reading *Basics*, consulting the *Topics* and *Objects* sections as references, and making use of Max's comprehensive on-line help.

Tutorial

This section is comprised of 40 chapters, designed to teach you about Max in a planned progression. We begin with the simplest concepts and build upon those concepts as new ones are introduced.

Each chapter is accompanied by a sample Max program (document) in the Tutorial folder. The sample program lets you demonstrate for yourself what's being explained in the text.

Topics

This section covers specific issues and areas of using Max, such as programming efficiency, what the menu commands do, graphics programming, how to write patch librarians, and so forth. Topics are arranged in alphabetical order, with each section identified by a large header.

Introduction

Objects

This section is organized in alphabetical order, and contains technical information on every object supplied with Max. Each object description is organized into five categories:

- *Input* describes each message the object accepts and what happens as a result
- *Arguments* specifies the arguments you type in when creating an object
- *Output* describes what comes out of the object's outlets
- *Examples* shows a couple of ways to use the object in a Max patch
- *See Also* refers you to descriptions of other related objects, relevant sections in *Topics*, or to *Tutorial* chapters which discuss the object

Hardware Requirements

To use Max you must have at least:

- A Macintosh Plus or higher with 1MB or more of memory. A hard disk drive and 2MB of memory is strongly recommended.

To use Max with MIDI you should have:

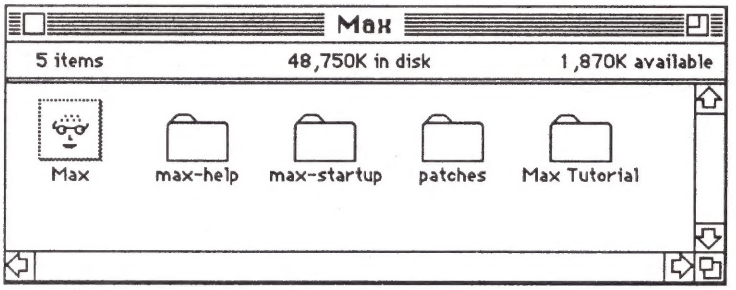
- A MIDI interface and MIDI cables to connect the computer and the MIDI equipment.
- A MIDI controller such as a keyboard.
- A MIDI-compatible sound source such as a synthesizer or a sampler. (The controller and the sound source may well be the same piece of equipment.)

Of course, you can use Max with much more MIDI equipment than described in this minimum setup.

Installation

Max is shipped on three disks. Disk 1 contains the Max application as well as the documents needed to use the MIDI Manager. Disk 2 contains four folders: *max-startup*, *max-help*, *patches*, and *Max Tutorial*. Disk 3 contains materials for developing Max objects in C. Non-C programmers can safely avoid handling Disk 3.

1. Create a folder on your hard disk, and name it something like *Max*.
2. Double-click on the Max icon on Disk 1. Click the INSTALL button. Choose the Max folder on your hard disk. Click the Install button again to begin the installation. Click QUIT. Copy the folders from Disk 1 into your Max folder.
3. Copy the folders from Disk 2 into your Max folder. The Max Tutorial folder may be removed after you have finished using the Tutorial to learn Max.



The MIDI Manager is not required to use Max, but there are a few things that are possible only if you are using the MIDI Manager.

4. If you do not already have the MIDI Manager installed, copy the MIDI Manager and the Apple MIDI Driver into your System Folder. You must restart the computer in order for the MIDI Manager to be available.

5. If you'll be using MultiFinder, copy the PatchBay application, which is used to configure the MIDI Manager, anywhere on your hard disk. Otherwise, install the PatchBay Desk Accessory into your system.

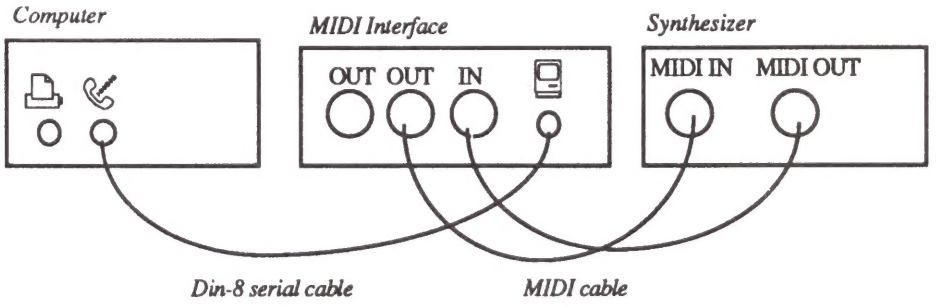
After you have copied the contents of the Max disks onto your hard disk, put the Max disks in a safe place in case you need to restore any files.

Connecting MIDI Equipment

1. Connect the MIDI interface to one or both of the serial ports of the Macintosh.

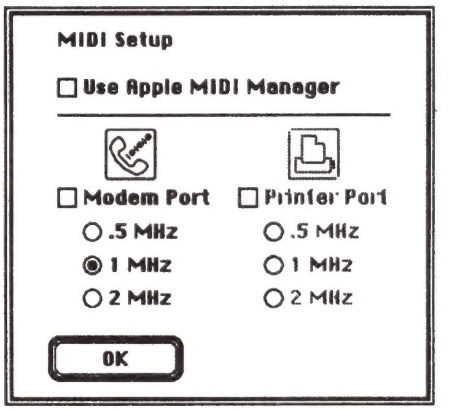
2. Connect the MIDI Out of the MIDI controller(s) to the MIDI In of the interface, in order to send in MIDI messages to Max.

3. Connect the MIDI Out of the interface to the MIDI In of the sound source(s), in order to send out MIDI messages from Max.



Configuring MIDI

The first time you open the Max application, you will be presented with the **Midi Setup...** dialog, for configuring the serial ports.



If *Use Apple MIDI Manager* is shown in black lettering, it means that the MIDI Manager is installed in your System. To use the MIDI Manager, check Use Apple MIDI Manager.

The MIDI Manager uses *virtual ports* to connect to Max. The actual connections between the Max application and the serial ports of the computer are specified in the PatchBay application. However, the default connections Max tries to make are the same as if you weren't using the MIDI Manager.

If you will not use the MIDI Manager, check either Modem Port or Printer Port, and select the appropriate interface speed (almost always 1MHz).

The **Midi Setup...** dialog box is presented to you automatically only the first time you use Max. From then on, Max remembers the specified setup (by storing it in the Max Preferences file in your System Folder). To change the configuration, choose **Midi Setup...** from the File menu.

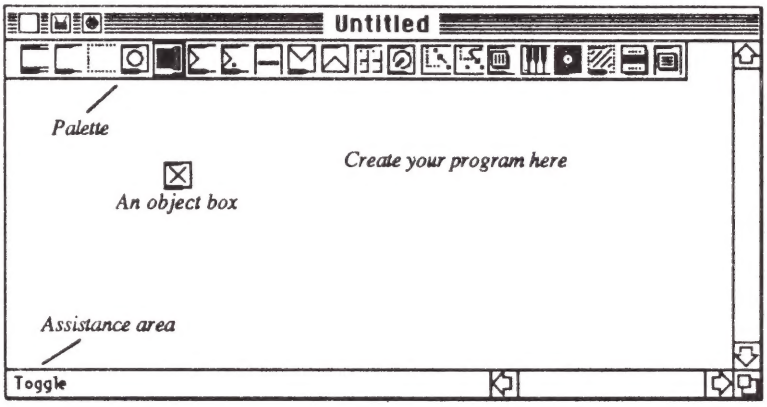
Programming With Objects

When you work with Max, you will be developing applications *graphically* using objects, which will appear on your screen as boxes that contain either text or icons. The boxes are connected together to create a working program.

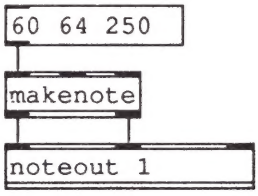
Max has about 100 *built-in* objects, each of which performs one or more specific tasks. In addition there are *external objects* that can be loaded into Max when it is running. Finally, you can create your own objects in Max itself, and use them inside other programs that you write.

Max Windows

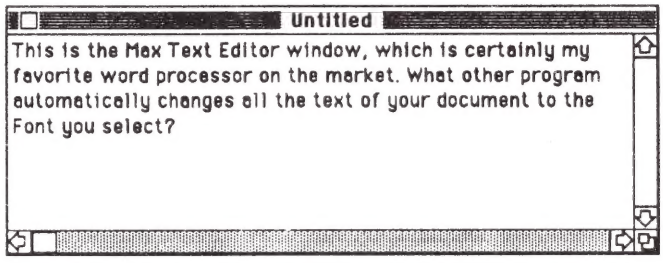
You create a Max program in a *Patcher* window which works much like a drawing program. You select an object from a palette, then click where you want the object to go.



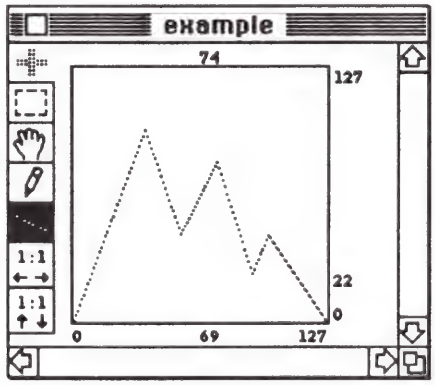
When you're finished with your program, you can save it as a Max document, otherwise known as a *patch*. Here's an example of a patch that plays Middle C for one-quarter of a second.



Max also has a simple Text window for taking notes or editing the data contained in certain objects.

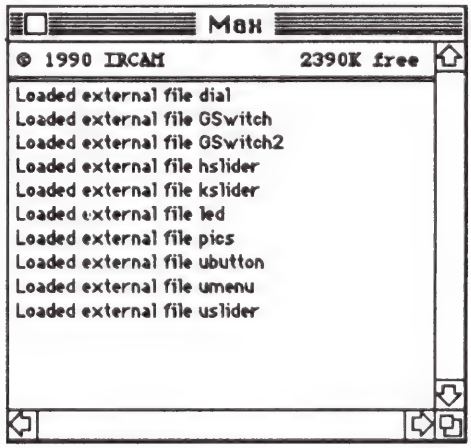

Untitled
This is the Max Text Editor window, which is certainly my favorite word processor on the market. What other program automatically changes all the text of your document to the Font you select?

Another window you'll see often is the *table editing window*, used to view and edit an array of numbers. The table window has a corresponding **table** object that you will use in a Patcher program to access the data you see in the table window.



The Max Window

When the Max application is first started, you'll the Max window first. This is the place Max prints out messages to the user. First, you'll see a list of any external objects which have been loaded, such as in the window shown below.



While you're using Max, any error messages or warnings generated while you are editing or running a program will appear in the Max window. You can also use the Max window in your own programs, which is especially helpful while you're learning Max or debugging your programs.

Objects

Objects are different types of boxes, like this:

Objects have *inlets* at the top, used to receive information from other objects...

...and *outlets* at the bottom, used to send information to other objects.

You connect two objects together with a patch cord. Always drag from the outlet of one object to the inlet of the other. When dragging a patch cord, you can be anywhere inside the box and the inlet will "expand" to indicate that you can release the mouse button to make the connection.

Inlet expanding to meet a patch cord

The most common type of object is the *Object box*, which has two lines at its top and bottom.

The actual action performed by an object box depends on the name typed into it. The name is like a verb, describing what the object will do. The name will either be a single word, such as **makenote**, or a symbol such as **+** (add) or **>** (is greater than).

Typically, objects send information out their outlets in response to information they receive in their inlets. Each inlet and outlet of an object has a different purpose. These are described in the *Objects* section of this manual. A brief description of the use of each inlet and outlet is available in the *Assistance* area of the patcher window. Just move the mouse over the inlet, and descriptive text appears in the Assistance area.

Getting Assistance About the Right Inlet of the **delay** *Object*

A few objects lack either inlets or outlets, because they receive or send information from somewhere outside of the patch that contains them. For example, the **midiout** object transmits values directly to MIDI.

Other objects are *user interface* objects. They look like buttons, dials, sliders, keyboards, etc., and respond not only to messages received from other objects, but also to clicking and dragging with the mouse. They let you control the program by using the mouse, and also provide a means of displaying numbers and other messages onscreen in a variety of ways. The name of each object is printed in the assistance area when you click on its icon in the palette.

There is one object, **comment**, that does *nothing*. It lets you put comments in a program—notes to yourself reminding you what you have done, or notes to others telling them how the program works.

Arguments

Often additional words or numbers are included in an object box after the object name to provide data to the object or to specify some of its characteristics. The additional words are known as *arguments* to the object. A few objects *require* arguments, while others have *optional* arguments.

The type of arguments which can be given to each object are detailed in the *Max Reference* section. If you forget to type an obligatory argument into an object box, Max will print an error message advising you of that fact, and will refuse to create the object.

Numbers

Max distinguishes integer numbers from decimal numbers (with a fractional part). Integer numbers are stored in Max in a data type called *int*, and decimal numbers are stored in a data type called *float*.

When a number is converted from float to int by Max, the fractional part of the float is chopped off (*truncated*) rather than rounded up. Thus, when the number 4.1 is converted to an int, it becomes 4. Likewise, the number 4.999999 becomes 4 when converted to an int.

The majority of the numbers you use in your programs will be ints. MIDI data is always expressed in terms of ints in Max, and since time is usually measured in milliseconds in Max (a millisecond is the smallest unit of time you can express in Max, even though the computer is working much faster than that), ints are used for milliseconds as well.

You can type in any int as a hexadecimal number. Hexadecimal is a base-16 number system often used in MIDI specifications and other computer-related documentation. Precede a hexadecimal number with 0x (zero plus a small x), as in 0xF0 (same as 240).

Messages

A *message* is the information passed from one object to another. A message can be a number, a *list* of numbers separated by spaces, a word (referred to in Max as a *symbol*), or a word followed by numbers and other words.

The first item in a message tells the receiving object what kind of a message it is. This item is known in object-oriented programming as the *selector*. Each symbol or number following the selector is called an *argument*.

The types of selectors for Max messages are:

int — When a message consists of nothing but an integer number (such as the message 127), its selector is automatically understood by Max to be int. It is not necessary for a message to say int 127.

float — When a message consists of nothing but a number containing a decimal point (such as the messages 3.97 or 3. or .97), its selector is understood by to be float. As with ints, it is not necessary to say float 3.97. The decimal point lets Max know it is a float.

list — A list of two or more numbers separated by spaces (such as 60 79 1.02 4) has the selector list. The message need not begin with the word list. Max recognizes a list whenever it sees a message consisting of a number followed by anything. In fact, a list can contain symbols as arguments (as in the message 1 start 768), so long as it *begins* with a number.

bang The message bang is a special message which has the meaning "do whatever it is you do." For example, when the **random** object receives the message bang, it sends a randomly chosen number out its outlet.

symbol Many symbols are meaningful selectors when received by certain objects. The exact symbols understood by each object are listed in the *Objects* section. For example, the **seq** object, a sequencer for recording MIDI performances, responds to messages received in its inlet such as start, stop, record, delay, and print. All of those messages would be meaningless to an object like * (multiply), which expects only numbers (or bang) in its inlets.

The word symbol itself can be used as a selector, followed by a single symbol as its argument (as in the message symbol ready). The symbol selector notifies the receiving object not to try and interpret the argument as a selector. In effect, it says, "Here comes a word, but it has no meaning." In this case, the word ready all by itself would be interpreted as a message by the receiving object, and would probably result in an error message.

When an object receives a message it doesn't understand, it will either ignore the message entirely or will print an error report in the Max window. In many cases, Max even stops you from making such mistakes *while* you are editing a program. As you connect an outlet of one object to the inlet of another object, Max does its best to analyze what message will be travelling through the patch cord. Max won't let you connect an outlet to an inlet that won't understand the message. If an inlet is not highlighted when you try to connect a patch cord to it, Max will not make the connection.

The **message box** displays and sends out the message it contains. It can contain anything you can type into it—a number, the word stop, even an entire sentence—up to 256 items. When it receives a mouse click, or a message in its inlet, it sends its own message out the outlet.

60 64 1
print
Key 60 with velocity 64 played on MIDI channel 1
print
60 64 1
Key $1 played on MIDI channel $3 with velocity $2
print

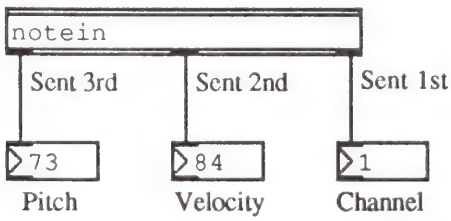
The message box can also contain *changeable arguments* which are replaced by arguments from the message it receives in its inlet. In the above example, the changeable argument $1 is replaced by the first argument in the incoming message, 60, the changeable argument $2 is replaced by the second argument received, 64, and the $3 argument is replaced by 1.

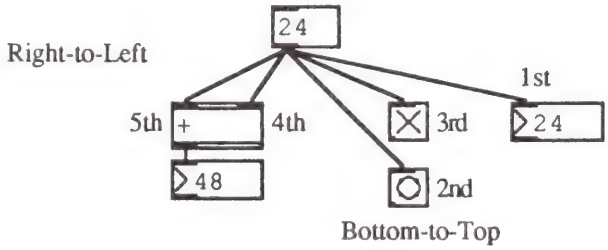
The **print** object can receive *any* message, but does not try to understand what it receives. It simply prints out the message verbatim in the Max window.

Order of Messages

An object often has more than one outlet, and usually sends messages out all outlets "at the same time." Actually, nothing *really* happens at the same time in Max. Things can happen so fast as to seem simultaneous, but it's important to know how Max orders messages.

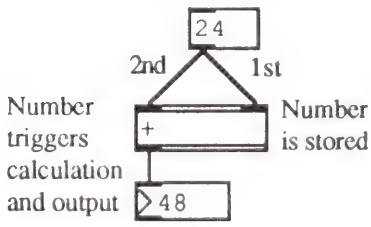
An object sending many messages at once, out different outlets, the order is *right-to-left*. The rightmost outlet sends its message out first, then the outlet just to the left of that, and so on until the leftmost outlet. This is true of virtually *every* object in Max.

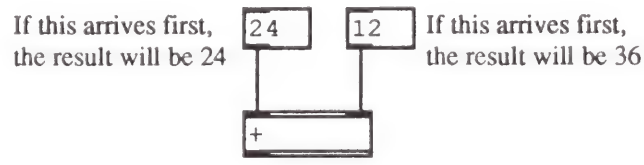




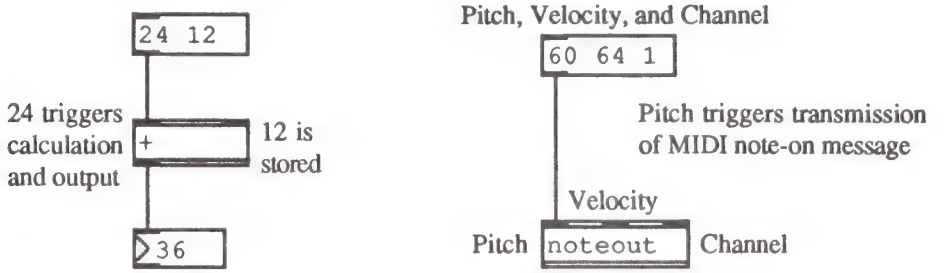
In the example shown at right, a single outlet has many patch cords connected to it, all leading to different inlets of other objects. In this case, the order that messages are sent is determined by the *right-to-left* screen position of the receiving objects. If two receiving objects are perfectly aligned vertically (neither is further right than the other), the order is *bottom-to-top*; the object that is lower on the screen receives the message first. If a single outlet is connected to two inlets of the same receiving object, the rightmost inlet will receive the message first.

When an object has more than one inlet, it *expects* to receive messages in its inlets in right-to-left order. Generally, an object will store all the messages it receives *until* it receives a message in its leftmost inlet, then it will perform some operation and send messages out its outlets. The vast majority of objects are "triggered" by a message in their leftmost inlet. There are some exceptions to this rule—notably the objects **timer**, **buddy**, **gate**, and **switch**—but the left inlet is the triggering inlet for almost all other objects. Notice that this makes sense because objects *send* their messages right-to-left, so the last message an object receives will usually be the one that triggers output.





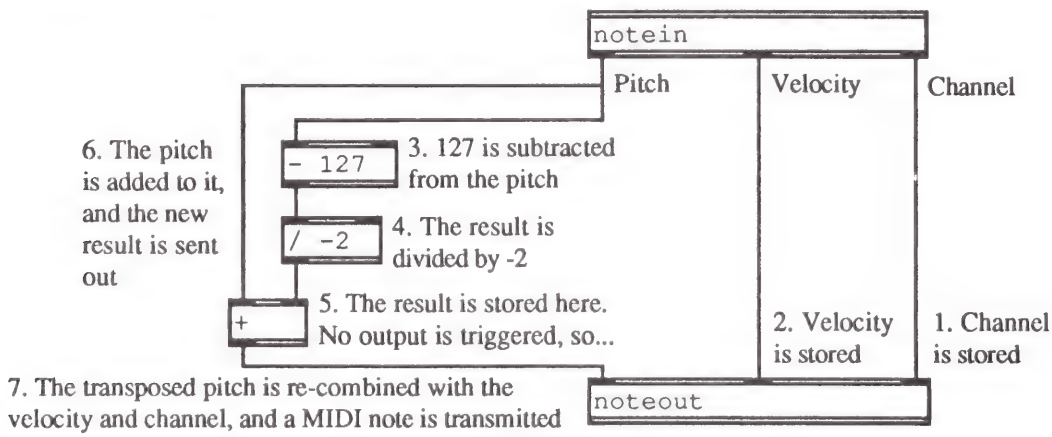
Many objects can receive a list of numbers in the left inlet, and the result will be the same as if the numbers had been received in different inlets in right-to-left order.



There are also objects whose sole purpose is to rearrange the order in which messages are sent to different objects. These objects are **swap** and **fswap** for reversing the sequential order of two numbers, **buddy** for receiving numbers in any order and then sending them on in a specific order, **trigger** for sending one number to many places in a specific order, and **bangbang** for sending one bang to many places in a specific order.

Finally, suppose that a message is sent several places from the same outlet, but that message triggers the receiving objects to send *their* messages. What is the order in that case? The answer is that the message will trigger the receiving object, and that object's output will be sent next. The messages will continue triggering other objects until a message is sent that triggers no other action, then Max goes back to the original sending object and sends its next message.

In the example below, note that steps 3, 4, and 5 are all performed (up to the point where no further message is triggered) before the message is sent to the left inlet of the + object.


notein
Pitch
Velocity
Channel
6. The pitch is added to it, and the new result is sent out
- 127
3. 127 is subtracted from the pitch
/ -2
4. The result is divided by -2
+
5. The result is stored here. No output is triggered, so...
2. Velocity is stored
1. Channel is stored
7. The transposed pitch is re-combined with the velocity and channel, and a MIDI note is transmitted
noteout

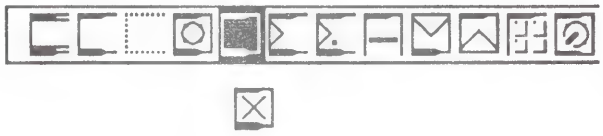
Editing and Running Programs

Programs are "written" in Max by placing objects in a Patcher window and connecting them with patch cords to make *patches*. Once you have created the patch, you can *run* it and ensure that it actually does what you want it to do. You can save the patch as a Max document, to run or edit

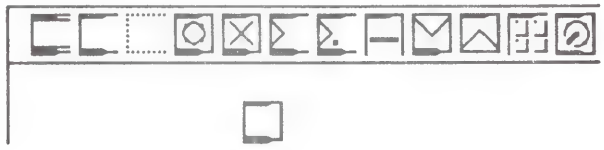
again later. A Patcher window is either *locked* or *unlocked*. When the window is unlocked, your patch can be edited, and you'll see the palette at the top of the window.

To place an object in the window, click on the desired icon in the palette.

The cursor will become the same as the palette icon.

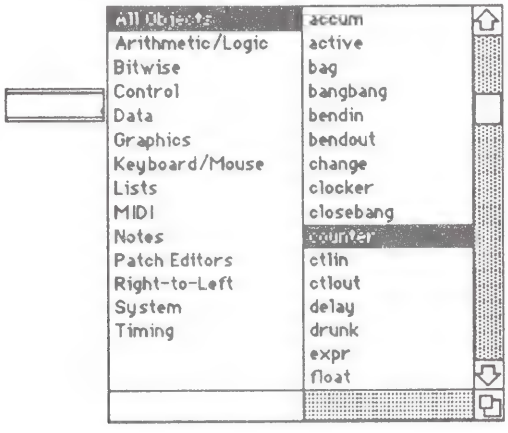


Then click in the Patcher window where you want to place the object.



New Object List

When you add an Object Box to a Patcher (the leftmost icon in the palette), you the New Object List window. The window is divided into two columns. The left column lists categories of objects, and the right column lists the names of objects within the selected category.



You can use the keyboard in the New Object List as you would into a standard Macintosh "open file" dialog. This can be a shortcut for typing the entire name of an object, especially when the current category is All Objects. When the object name you want is selected, double-click, or press Return, Enter, or Space, and the name will be entered into the object box. If you don't want any of the names shown in the New Object List, press the Backspace key, and the window will go away without entering any text into your box.

Editing Operations

When you click on a box, it is selected for editing operations, such Copy, Duplicate, or if the box contains text, changing its Font or Font Size. You can also drag the box to a new location. You can resize a box by clicking on its grow bar. The cursor will change to a two-way or four-way arrow when you are above the grow bar (which is sometimes invisible).

When the window is locked, the object palette disappears, and clicking on boxes operates them as parts of the user interface. There are several ways to lock and unlock a patcher:

• Click on the lock icon in the title bar of the Patcher window.

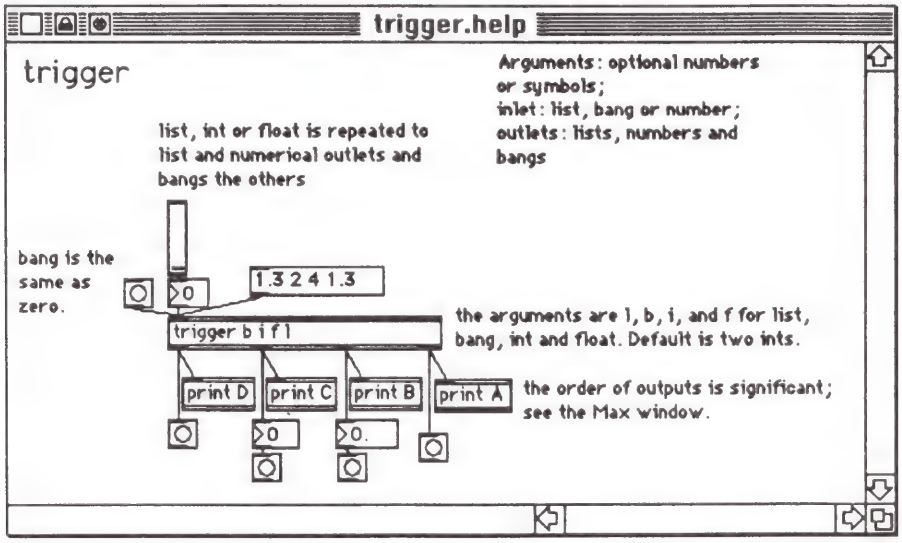
• Choose **Edit** from the Max menu, or type Cmd-E.

• Command-click on the "white space" in the Patcher window.

You can also command click on any object when the Patcher is unlocked to operate it as though the Patcher were locked. This is a very handy shortcut to know.

Help

You can access help files for built-in and external Max objects at any time. Option-click on an object opens the Help file for that object. Help files are actual operating Patcher files that let you see an object "in action." Show below is the help window opened for the **trigger** object.


trigger.help
trigger
Arguments: optional numbers or symbols;
inlet: list, bang or number;
outlets: lists, numbers and bangs
list, int or float is repeated to list and numerical outlets and bangs the others
bang is the same as zero.
0
1.3 2 4 1.3
trigger b i f l
the arguments are l, b, i, and f for list, bang, int and float. Default is two ints.
print D
print C
print B
print A
the order of outputs is significant; see the Max window.
0
0.

Choosing **Help...** from the Max menu when no object is selected will bring up a general help file.

Max Preferences

Max remembers many of your working preferences by storing settings each time you quit the application. The first time you use Max, a Max Preferences file is created in your System Folder, where Max stores settings you have chosen for in the MIDI Setup and File Preferences dialogs, as well as the current settings of the Options menu. In addition, you can set a default font and font size for the Max window and subsequent new Patcher windows by using the Font menu while the Max window is the active window.

Sending and Receiving MIDI

Max has many built-in objects for sending and receiving MIDI data to and from the serial ports or the MIDI Manager. Objects which receive MIDI messages from "the outside world" have no inlets, since their input comes from the serial ports or from the MIDI Manager, rather than from other Max objects. Likewise, objects which transmit MIDI messages have no outlets, since they send MIDI data.

The **midiin** and **midiout** objects receive and send raw MIDI data one byte at a time, without analyzing the MIDI messages at all. These objects can then be connected to objects which record the MIDI data, process it, or play it back.

You will probably end up using the specialized MIDI objects the most. They filter the raw MIDI data coming into Max, and output only the vital information. For example, the **notein** object looks only for MIDI note-on messages, and when it receives what it is looking for, it outputs the key number, the velocity, and the channel number. Similarly, the **bendin** object looks only for incoming pitch bend messages, and sends out the amount of pitch bend and the channel number. Other specialized MIDI input objects are **ctlin** for control change data, **polyin** for polyphonic key pressure data, **touchin** for aftertouch data, **pgmin** for program change data, **rtin** for real time messages, and **sysexin** for system exclusive data.

Each of the specialized MIDI objects that receives channel voice messages has a transmitting counterpart, which receives numbers from other Max objects and uses those numbers as data for transmitting specific MIDI channel voice messages. The specialized MIDI transmitting objects are: **noteout**, **polyout**, **ctlout**, **pgmout**, **touchout**, and **bendout**.

Each Patcher window has a MIDI Enable/Disable button — — in its title bar. When the button shows a MIDI plug icon, the MIDI objects are enabled and function normally. When the button shows an X, all MIDI receiving and transmitting objects in that window are disabled. Clicking on the button with the mouse toggles it back an forth between Enabled and Disabled.

Processing MIDI

Once MIDI data has been received by the MIDI objects, it can be modified and manipulated by other objects in Max. Max has many objects designed specifically to handle MIDI data.

The **midiparse** object receives numbers from **midiin**, analyzes the type of MIDI message, and routes MIDI channel voice data out specific outlets. The **seq** object is a simple MIDI sequencer that records a sequence of data received from **midiin**, and can play it back later, even changing the speed. A multi-track sequencing object, **mtr**, can record many separate tracks of data, then play them back either simultaneously or individually. To format numbers into well-formed MIDI messages, there is a **midiformat** object that prepares MIDI messages, then sends them to **midiout** to be transmitted out from Max.

Borax analyzes note-on data received from **notein**, and sends out information regarding the duration of notes, the time between note-ons, how many notes are being held at any given moment, how many notes have been played, etc. Both **Borax** and the **poly** object are capable of assigning voice

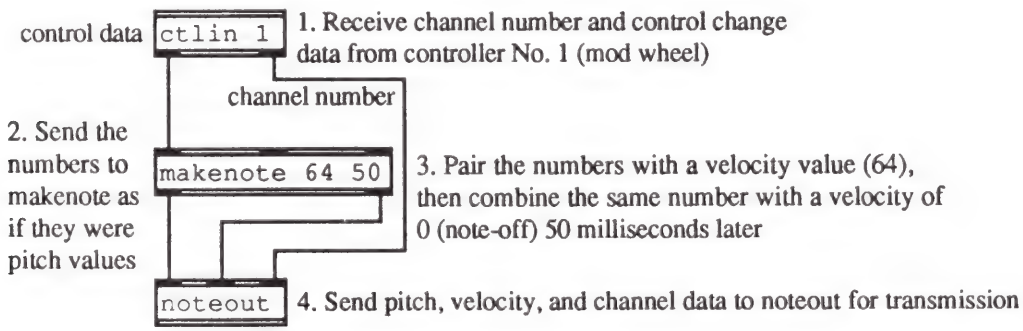
numbers to the notes that pass through them, and **poly** can steal voices and provide note-off messages to turn off notes that are being held.

The **stripnote** object filters out note-off messages (note-on messages with a velocity of 0) received from from **notein**, passing on only note-on messages with a positive velocity. The **makenote** object is like the reverse of **stripnote**. It adds note-offs after note-on messages generated within Max, then sends both the note-ons and the note-offs to **noteout** for transmission.

Max usually uses only 7-bit values (numbers from 0 to 127) for pitch bend data, and uses note-on messages with a velocity of 0 to express a note-off. It is capable, however, of recognizing and transmitting 14-bit precision pitch bend values and MIDI note-off messages using **xbendin**, **xbendout**, **xnotein**, and **xnoteout**.

The above objects are specifically designed to handle MIDI data, but since all MIDI values are just expressed as ints in Max, they can be processed in any way imaginable. The **pipe** object, for instance, delays all numbers passing through it by a certain amount of time. Or the **+** object can be used to add some number to a MIDI pitch value, thus transposing the pitch. The other sections in this manual will show you a few of the many ways to process numbers in Max.

MIDI data can be reassigned to have other meanings. For example, data from the mod wheel of the MIDI keyboard could be received with **ctlin**, sent to **makenote** (to combine the number with a velocity value and add a note-off after it), then transmitted by **noteout**, as shown below. This would let you play *notes* with the mod wheel.

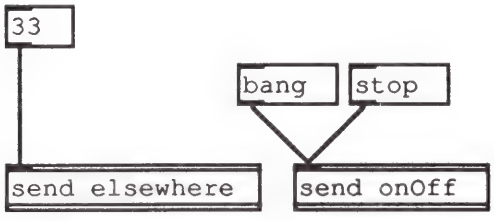

control data
ctlin 1
1. Receive channel number and control change data from controller No. 1 (mod wheel)
channel number
2. Send the numbers to makenote as if they were pitch values
makenote 64 50
3. Pair the numbers with a velocity value (64), then combine the same number with a velocity of 0 (note-off) 50 milliseconds later
noteout
4. Send pitch, velocity, and channel data to noteout for transmission

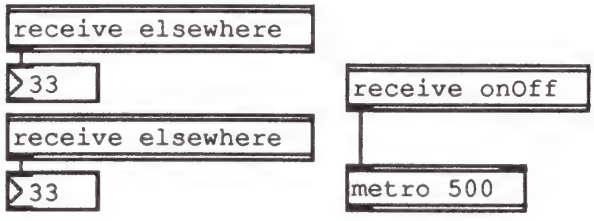
Numbers can also be generated *within* Max, either automatically with timing objects such as **metro**, **tempo**, and **clocker**, or interactively onscreen with user interface objects such as **slider**, **dial**, and **Number box**. The numbers generated in Max can be sent to MIDI objects and transmitted out to play music.

send and receive

Once you start writing complex programs with Max, you'll begin to feel restricted by having to put an entire program in a single Patcher window. There are many ways for a program to be divided up between numerous Patcher windows, and ways for several windows to communicate with each other.

The **send** and **receive** communicate with each other without patch cords. Each **send** and **receive** object has one argument, a symbol that it uses as its *name*. Any message that goes into the inlet of a **send** object, immediately comes out the outlet of *all* **receive** objects which share the same name. In effect, **send** and **receive** objects with the same name are like radio transmitters and receivers which are tuned to the same frequency. These objects *need not be in the same window* to communicate, so long as they have the same name.



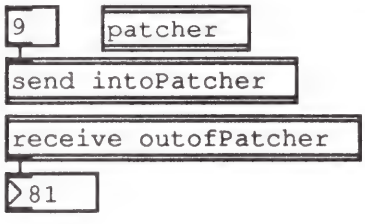


Any single word can be used as a name for **send** and **receive** objects, so long as it doesn't contain special characters such as a comma or a semicolon, and doesn't begin with a number.

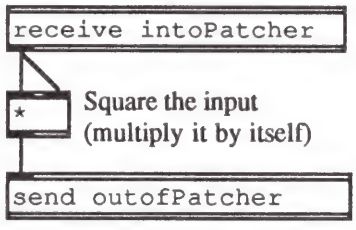
Another object, **value**, allows a single number value to be shared by all **value** objects which have the same name, even if they are in different patches. When the number in one of the **value** objects is changed, the number is changed in *all* **value** objects which have that name.

Object Named "patcher"

An entire Patcher program can be *embedded* in another Patcher program, as a *subpatch*. An object called **patcher** actually opens a new Patcher window, in which you can write another patch (a *subpatch*), and that subpatch will be saved as part of the Patcher file that contains the **patcher** object. A patch can contain one or more **patcher** objects, and subpatches can be nested to any degree. One way to communicate between a patch and a subpatch is to use **send** and **receive** objects, as shown in the example below.



The contents of the patcher object look like this:



Inlets and Outlets

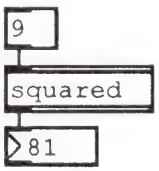
You can give a **patcher** object its own inlets and outlets by placing **inlet** and **outlet** objects inside the subpatch. In this way, messages can be sent to the subpatch through patch cords. The messages received in the inlets of the **patcher** object come out of the **inlet** object inside the subpatch. Similarly, messages received by the **outlet** object in the subpatch come out the outlets of the **patcher** object.

The contents of the subpatch look like this:

Multiply the number times itself

Your Object

Once a patch has been saved as a Max document, it can be used as an object in its own right. Just create a new object box and type in the filename of your saved patch, and the saved patch will be loaded into the patch you are editing, just like any other object. The patch you created earlier and saved as a document should usually have **inlet** and **outlet** objects in it (or **send** and **receive** objects) so that it can communicate with the patch that uses it as an object.



The contents of the "squared" object look like this:

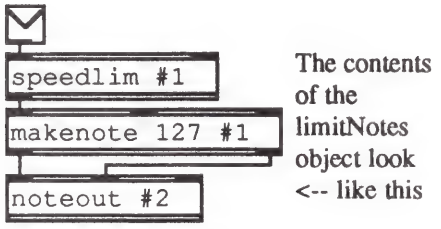
Multiply the number times itself

"squared" is not really a Max object; it's a patch that has been saved as a document

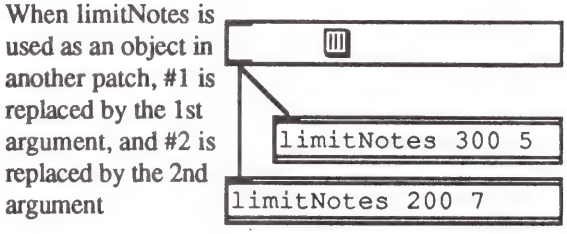
It is easy to see that many different patches can be created separately, then communicate with each other when they are used as individual objects in one *main* patch.

Your Argument

In the same way that many Max objects require you to type in arguments to give them information, your own objects can require typed in arguments to supply them with information. This is done by including a pound sign and a number (for example, #1) in the patch you plan to use as an object. A symbol such as #1 is a *changeable argument*, and is replaced by whatever number or symbol you type in as the corresponding argument when you use the patch as an object inside another patch. A changeable argument cannot be used to supply the name of an object itself, but can be used as an argument anywhere inside your object.



Built-in Max objects


When limitNotes is used as an object in another patch, #1 is replaced by the 1st argument, and #2 is replaced by the 2nd argument
limitNotes 300 5
limitNotes 200 7

limitNotes is a patch saved as a document

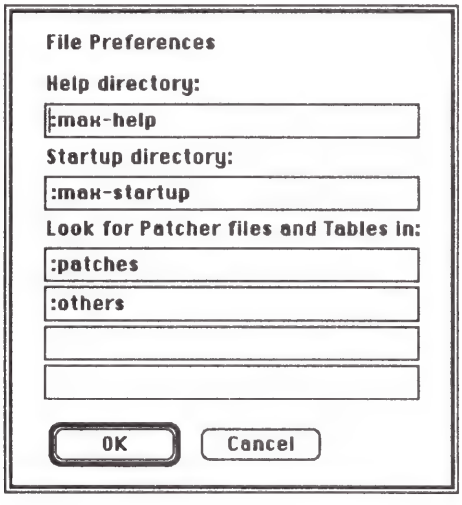
In the above example, the arguments typed into the **limitNotes** object boxes supply values to the objects inside **limitNotes**. Notes are played with the horizontal slider (**hslider**). When the **hslider** is moved, one **limitNotes** object plays a note every 300 milliseconds on MIDI channel 5, and the other plays a note every 200ms on MIDI channel 7.

Important: A patch that has been saved as a document must never be *recursive*—that is, it must not contain *itself* as an object. For example, **limitNotes** must not contain a **limitNotes** object. If it did, Max would get into an infinite loop of trying to load objects within objects within objects, and so on.

File Preferences

In order to keep track of help files, external objects, and Max objects you've created yourself, Max lets you define the names of folders that it should use in looking for various files when you choose **File Preferences...** from the Edit menu.

The File Preferences dialog box, shown below, lets you change name of the Help directory (usually *:max-help*), the Startup directory (usually *:max-startup*, where external objects are loaded at startup), and four other folders to look for other files. If a directory name starts with a colon, it is a subdirectory of the folder that contains the Max application. Otherwise, the name must start with the name of the volume (disk drive) which contains the folder.



By default, Max will look for patcher files, external objects, tables, and other files in a folder called *:patches*. You can specify additional folders by adding their names below *:patches*. One suggestion for using this feature is to place all the external objects which are user interface objects in the startup folder, the non user interface external objects in another folder listed in the File Preferences dialog, and your reusable Max objects in *:patches*. Refer to *Objects* in the Topics section of the manual, which explains how to add the names of objects that are not loaded into Max in the New Object List. Not loading all your external objects at startup will save a few seconds each time you run the program, and can also cut down on the amount of memory Max uses.

Tutorial

Introduction

This *Tutorial* is a step-by-step course designed to teach you all about Max, beginning with the simplest concepts and building upon those concepts as you learn new ones. The course is primarily for new Max users who don't have prior programming experience, but even if you have some knowledge of programming, the Tutorial is a good way to learn Max.

The Tutorial consists of 40 chapters, designed to be read in order. Each chapter is accompanied by a sample Max program (document) in the *Max Tutorial* folder. The document is a working illustration of the concepts in the chapter text—it lets you see Max in action and try things yourself. We feel this hands-on approach is a more efficient way to learn about Max than just reading the manual by itself.

By the time you have completed the Tutorial, you will have a good understanding of Max and its capabilities, and will probably also have many ideas for your own Max applications.

As you read each chapter of the Tutorial, you will open the corresponding Max document in the Tutorial folder. The Tutorial even includes a few "quizzes" so you can be sure you understand the material before proceeding.

MIDI Equipment

The first few chapters of the Tutorial do not deal with MIDI directly, but simply teach you about some of the elements of Max. Later chapters do involve MIDI quite extensively, though, and in the sample programs we make certain assumptions about what MIDI equipment you are using and how it is connected to the computer. In order to benefit the most from the Tutorial, keep in mind these assumptions:

1. You are using a polyphonic synthesizer or sampler and a 61-key velocity-sensitive keyboard with pitch bend and modulation wheels. Your keyboard should ideally be set to send on MIDI channel 1, and the snynthesizer set to receive in Omni On mode.

2. You have connected the MIDI Out of your MIDI keyboard to the MIDI In of your MIDI interface, and connected the MIDI Out of your interface to the MIDI In of your synth.

3. You have connected the MIDI interface to the modem port of your Mac.

Even if your equipment doesn't exactly match that assumed by the Tutorial, try to emulate the assumed setup as much as possible. You may even want to read the user's manual of your synthesizer, to be sure you understand its MIDI capabilities.

Tutorial 1

Saying "Hello!"

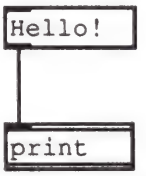
Open Tutorial 1

- If you have not already started up the Max application, do so now by double-clicking on the Max application icon in the Finder.

When you start the application for the first time, you will be presented with the **Midi Setup...** dialog box, for specifying the ports through which Max will be able to receive and transmit MIDI data. For the purpose of this Tutorial, your MIDI interface should be connected to the modem port of your computer. Check the box next to *Modem Port*, and specify the type of interface you are using. (*1 MHz* is the most common MIDI interface type.)

To open the sample program for each chapter of the Tutorial, choose **Open...** from the File menu and find the document in the Max Tutorial folder with the same number as the chapter you are reading.

- Open the file called *1. Hello!*



Objects and Messages

- Click in the box marked Hello!. Notice what happens in the Max window each time you click on Hello!.

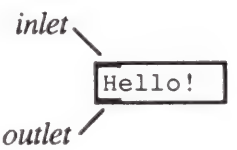
The basic operation of a Patcher program is simple. Different types of boxes, called *objects*, send *messages* to each other through *patch cords*.

This program contains two different objects:

The box containing the word print is a **print** object. A **print** object prints whatever message it receives in the Max window.

The word Hello! is a *message* contained in a **message box**, which can contain anything that can be typed. Often it will contain numbers.

Different kinds of Patcher objects have different numbers of *inlets* and *outlets*. The message box always has one inlet and one outlet.



Inlets are always at the top, indicated by blackened areas at the top of an object. Outlets are always at the bottom of an object.

The **print** object has no outlet—its output is always just printed in the Max window. Usually, an object will have *both* inlets and outlets; it receives messages, performs some task, then sends out messages. The **print** object just prints whatever it receives.

The message box object is connected to the **print** object by means of a *patch cord.* Just like components of a stereo system, the outlet of one object is connected to the inlet of another object. You can't connect an inlet to another inlet, or an outlet to another outlet.

The program operates as follows:

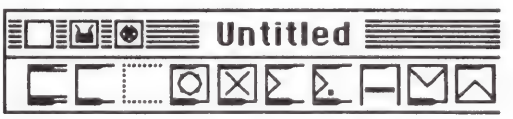
1. When you click on the message box object, the message Hello! is sent out the message box's outlet and through the patch cord.

2. The message reaches the inlet of the **print** object, which prints the message print: Hello! in the Max window.

Locking and Unlocking a Patcher Window

A Patcher window can be in one of two states: *locked* or *unlocked.* When a Patcher window is *locked,* it is a program ready to run. The locked state is shown by the closed padlock in the title bar of the window.

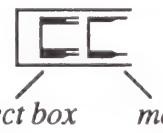
When a Patcher window is *unlocked,* you see a *palette* at the top of the window. An unlocked window is in **Edit** mode, and modifications can be made to the patch.



You can switch between a locked and unlocked Patcher window by clicking on the padlock, or by holding the Command key down and clicking in the window. Choosing **Edit** from the Max menu has the same effect.

- Click on the padlock and you will see the object palette. You can now modify the program.

The first two items in the palette are the object box and the message box.



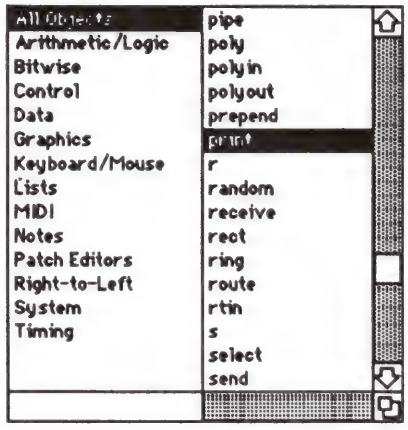
Tutorial 1

Saying "Hello!"

Modifying the Patch

Now we'll produce a program that prints "Good-bye!"

- Click on the object box in the palette. The cursor turns into an object icon. Click inside the Patcher window, near the bottom-right corner. A list of pre-defined Max objects called the *New Object List* will appear.



- Scroll down through the right-hand column of the New Object List until you see **print**, click on the word to select it, and type the Return key (or the Enter key). Alternatively, you could just type the first few letters of the word "print" until it is selected in the list.

  Note: If you want to type in the name of an object without using the New Object List, type the Delete (Backspace) key or click anywhere outside the New Object List and it will go away.

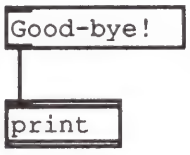
- You now have an object box with the word print in it. Click anywhere else in the window (outside that object box), and a **print** object is created with an inlet at the top.

- Next, click on the message box icon in the palette, and click just *above* your **print** object, to place a new message box in the window. Type Good-bye! into your message box.

To connect the message box to the **print** object, drag from the outlet of the message box to any place inside the **print** object:

- Position the cursor on the outlet of your message box. When the cursor is over an outlet, the outlet expands. Click on the expanded outlet and drag until the cursor is inside your **print** object and you see the *inlet* of the **print** object expand. Then release the mouse button. This will create a patch cord connection between the two objects.

Tutorial 1



If your message box and your **print** object are not perfectly aligned vertically, the patch cord will appear jagged. This has no effect on the functioning of the patch. However, if you're a fastidious person and want to clean up the appearance of your patch, select both objects just as you'd select multiple icons in the Finder (by Shift-clicking on each of them or by dragging across both of them with the mouse). Then choose **Align** from the Max menu.

You can also move objects by dragging them to the desired location. Objects and patch cords can be removed entirely by clicking on them to select them, then pressing the Delete key or choosing **Cut** or **Clear** from the Edit menu.

- Click on the padlock to lock the Patcher window. Your program is now ready to run. Click on the message box containing Good-bye! and you should see print: Good-bye! in the Max window.

Summary

When a Patcher window is *unlocked*, it is in **Edit** mode, and can be modified. When the window is *locked*, the program is ready to run. You can also run the program by holding down the Command key and clicking in the Patcher window.

A *message* is sent through a *patch cord* from the *outlet* of one object to the *inlet* of another. A message box contains any message you type into it. When you click on a message box, it sends its message out the outlet. A **print** object prints in the Max window whatever message it receives in its inlet.

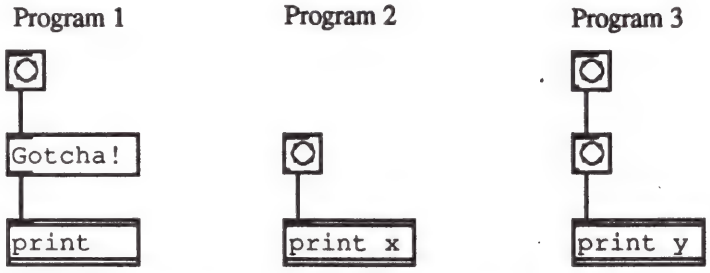
See Also

message	Send any message
Objects	Creating a new object
print	Print any message in the Max window

Tutorial 2

bang *means "Do It!"*

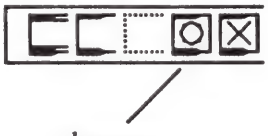
The bang Message



- This program is actually three separate printing programs. Click on the **button** icons in each program and notice what gets printed in the Max window.

The first thing to observe is that two of the **print** objects have names: x and y. Since there can be any number of **print** objects in a Patcher window, you will often want to make it clear which one is actually printing a message. You do this by putting a name after the word **print** in the box. When there's no name, the message is preceded by print:, as in Program 1. When there is a name, it precedes the message, as in Programs 2 and 3.

The second important new thing in this window is the **button** object. It appears as a separate item in the palette, and is really very much like a message box that contains the message bang.



button

You see, bang is a magic word in Patcher. It's a special message that means, "Do it!", which causes an object to do whatever it's supposed to do. For example, a message box sends out the message it contains in response to a bang or a mouse click.

- In Program 1, you can click on the "Gotcha!" message box to print it, or you can click on the **button**, which sends a bang message to the inlet of the message box. The effect is equivalent, since in either case the message box is "triggered" and sends out the message it contains.

- Program 2 not only proves that the **button** quite literally sends the message bang; it also proves that bang has no special effect on the **print** object. That's because the **print** object doesn't try to understand the message it receives. Its only purpose in life is to print out what arrives in its inlet.

- Program 3 is sort of a puzzler. Clicking on either **button** produces a printout of y: bang.

When you click on the upper **button**, which **button** actually supplies the message to the **print** object?

The answer is the lower **button**.

bang *means*
"Do It!"

Tutorial 2

The upper **button** sends a bang message to the lower **button**. The lower **button** interprets the bang message as "Do it!", and performs its expected function, which is to send a bang message. The **print** object simply prints out what it receives.

Summary

The message bang is a special triggering message which causes an object to perform its task. The **button** object's task is to send out the message bang, thus triggering *other* objects.

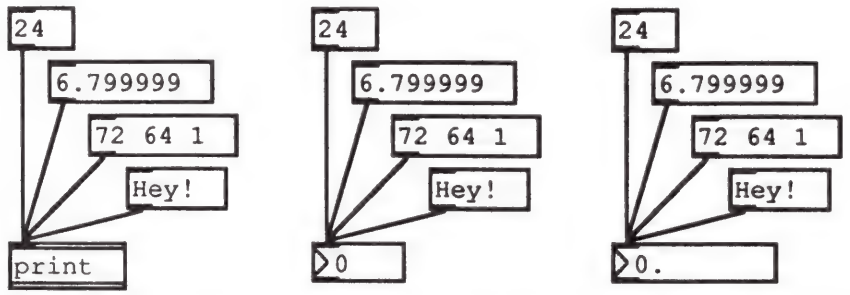
See Also

button	Flash on any message, send a bang

Tutorial 3

About numbers

int, float, and list



We have seen that a message can consist of text, and that some words have a special meaning to certain objects, such as the word bang. Very commonly, a message will consist of one or more numbers.

Max distinguishes integer numbers from decimal numbers (with a fractional part). Integer numbers are stored in Max in a data type called *int*, and decimal numbers are stored in a data type called *float*. Numbers you use in your programs will usually be ints.

Most of the time you won't really need to worry about this distinction in how numbers are stored, because Max will take care of it for you, and will even convert an int into a float or vice versa if it needs to (for instance, if a float is received by an object that expects to receive an int). The main thing you need to know is that when a float is converted to an int, its fractional part is not rounded off, but is *truncated*. (The fractional part is just chopped off.) For example, the number 6.799999 does not become 7, it becomes 6.

A message can also consist of several numbers, separated by spaces, which are all sent together. This is known as a *list*. A list can consist of both ints and floats. You'll encounter lists in later chapters of the Tutorial.

Number box

If you want to show a number in a Patcher window, use a **Number box**. There are two Number box objects in the object palette, one for showing ints and one for showing floats.

Number box (int) *Number box (float)*

A number received in the inlet of a Number box is displayed, and passed on out the outlet. This is an effective object to use as a "wiretap" to see what is the most recent number to have passed through a patch cord.

- Click on the different message boxes, and notice what is displayed, either in the Number boxes or in the Max window.

Notice a couple of important differences between printing messages with a **print** object, and displaying them with a Number box.

1. The **print** object will print any message it receives, regardless of the content of the message. The Number box, on the other hand, can display only one number at a time. If it receives a list, it displays (and passes on) only the first number in the list. If it receives an arbitrary text message, it does nothing except complain that it doesn't understand that message.

2. A Number box can show only an int or a float. If an int Number box receives a float, it converts the number to int, and vice versa.

The Number box has other features not described here. This patch does show one of its most common uses, though—to display the number that has most recently passed through a patch cord. You can learn more in the *Number box* Tutorial.

Summary

A Max *message* can consist of a single number, of type *int* (for integers) or *float* (for decimals). Most numbers used in Max (such as MIDI data and millisecond time values) are ints. A message can also consist of a space-separated *list* of numbers, which are all sent together in one message.

A Number box shows the most recent number it has received, and passes that number on out the outlet. A Number box is either of type *int* or *float*, and will convert numbers to that type.

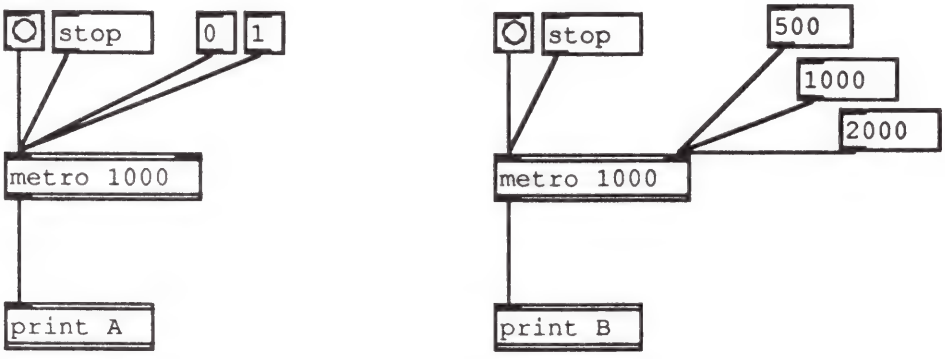
See Also

Number box Display and output a number

Tutorial 4

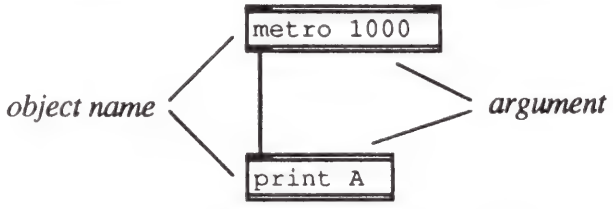
Using **metro**

Object Names and Arguments



In this chapter, we introduce a new object called **metro**, which functions as a metronome. You will notice that we have typed in a number after the word **metro** in the object box. This is the number of milliseconds between ticks of the metronome.

The number after the word **metro** is called an *argument*. We have already seen arguments used to give names to **print** objects. Arguments typically give objects information necessary to do their job.



Some objects require typed-in arguments in order to function. More commonly, an argument is optional, to supply some starting value, as in the case of **metro**, where the argument determines the initial speed of the metronome. When **metro** is started, it sends out a bang message every *n* milliseconds (where *n* is the argument) until the metronome is stopped. If no argument is typed in, metro has a *default* value of 5, and sends out a bang every 5 milliseconds.

The **metro** object has two inlets. A message received in the left inlet can start or stop the metronome. The metronome will start when it receives any any non-zero number in its left inlet, and it will stop when it receives a 0. Alternatively, you can send it a bang message to start, and stop to stop. A number received in the right inlet will change the number of milliseconds between bang outputs that was initially set by the argument.

- Try turning the **metro** objects on and off, and watch what is printed in the Max window.
- Try sending different numbers to the right inlet of **metro**, and notice the change in the speed with which messages are printed. The speed can be changed while the metronome is running, but the change does not take effect until the next bang is sent out.

Because it sends out the message bang, **metro** is a useful object for triggering other objects repeatedly at a specific speed.

Using
metro

Tutorial 4

Summary

After you type a name into an object box, you can supply additional information by typing in *arguments* after the object name. Arguments are usually *optional*, but some objects have *obligatory* arguments. If optional arguments are not typed in, Max usually supplies a *default* value.

A **metro** sends out bang messages repeatedly at regular intervals of time, until it is stopped. The number of milliseconds between bangs is specified by the argument or by a number received in the right inlet.

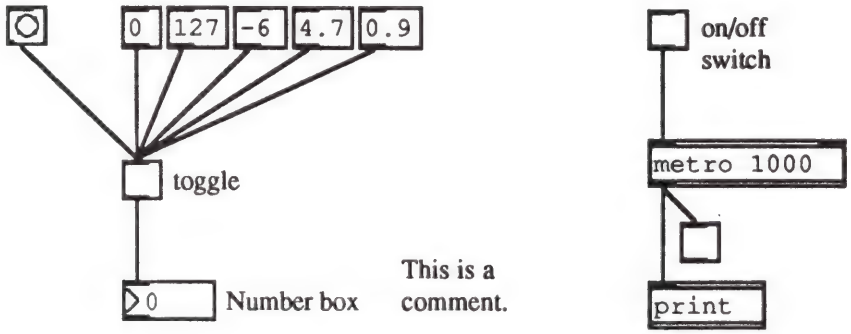
See Also

metro	Output bang, at regular intervals

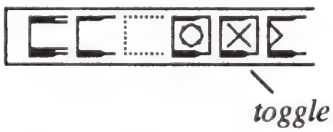
Tutorial 5

toggle *and* **comment**

toggle



The **toggle** object is the box with an X in it in the object palette. It functions as an indicator or a switch between two states: zero and non-zero.



- Click on the different message boxes containing numbers, and notice what happens to the **toggle** and the **Number box**.

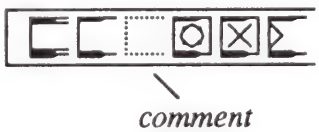
The **toggle** object can receive a number or a bang in its inlet. If the number is non-zero, **toggle** will show an X and send out the number. If the number is 0, the box will be blank and 0 is sent out the outlet. The **toggle** expects to receive an int, so when it receives a float it converts it to int. That is why the number 0.9 is understood as 0 by **toggle**.

The **toggle** alternately sends out the values 1 and 0 each time it is clicked with the mouse or receives a bang in its inlet. When it receives a bang or a mouse click, it reverses its state and sends out the new value. This distinction between zero and non-zero is Max's way of turning things on and off, or distinguishing between true and false.

- Thus, you can use a **toggle** as an on/off switch. In our example, the **metro** object can be turned on and off by clicking on the **toggle**. Try it. This works because **metro** starts when it receives a non-zero number (like 1) and stops when it receives a 0.

comment

The dotted box in the palette, to the right of the message box, is a **comment**.



toggle *and*
comment

Tutorial 5

A **comment** has no effect on the functioning of a program. It's simply a way of putting text into a Patcher window. The main reasons to use **comments** are:

1. To label objects in the patch, such as "on/off switch".
2. To give instructions to the user, such as "Click here".
3. To explain the way a program works, or how a particular item in a program functions. This is not only helpful to the user of the program, but is also very helpful to you, the programmer. You'd be amazed how quickly you can forget how your own program works. Get in the habit of adding many explanatory **comments** as you build programs.

A **comment** box (or virtually any other object) can be resized by dragging on the *grow bar* in the lower-right corner of the box.

- You can also change the size of the text in a **comment** (or any other object). Click on the **comment** box to select it, then choose a different font or size from the Font menu. Try changing the font characteristics and the size of the **comment** that says "This is a comment."

When you specify font characteristics with *no* objects selected, you set the characteristics for any new objects you subsequently create in the active window. When you specify font characteristics with the Max window in the foreground, you set the characteristics for *all* new Patchers you subsequently create. Max stores these font characteristics in the Max Preferences file in your system folder, and recalls them each time you use Max.

Summary

A **toggle** can be used to generate the numbers 1 and 0, for turning other objects (such as **metro**) on and off. It can also be used as an indicator of numbers passing through it, telling whether the most recent number was zero or non-zero (although any floats passing through will be converted to int.) A **comment** doesn't do anything, but is useful for putting text in a Patcher window.

See Also

comment	Put explanatory notes or labels in a patch
led	Display on/off status, in color
TogEdge	Report a change in zero/non-zero values
toggle	Switch between on and off (1 and 0)
ubutton	Transparent button, sends a bang

Tutorial 6

Test 1: Printing

Make a Printing Program

Here is an exercise to make sure that you understand what has been explained so far.

- Create a Patcher program which, when turned on, prints the phrase...

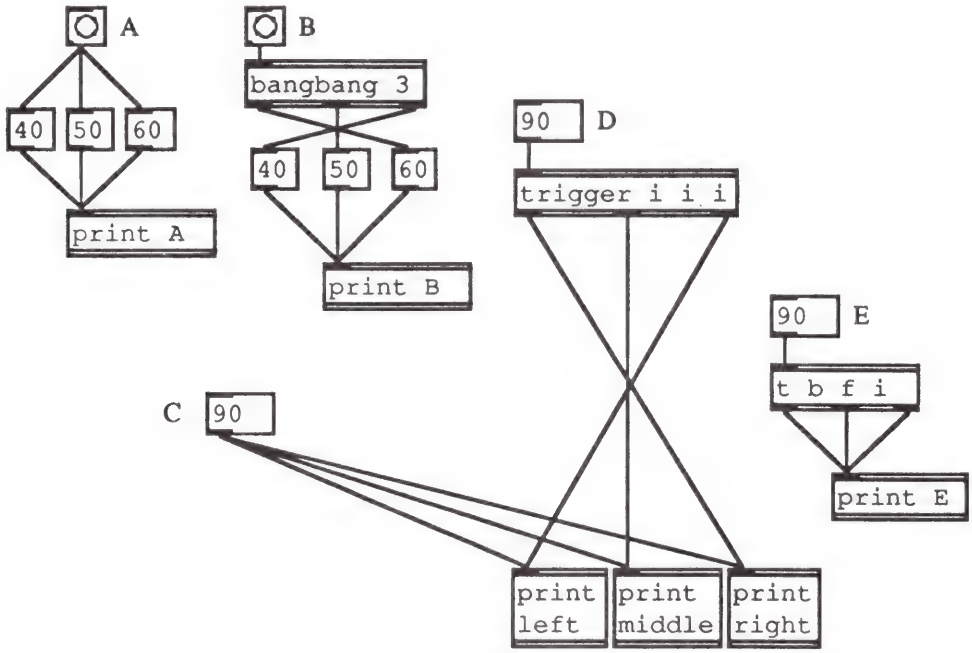
  test: 1

  ...in the Max window every two seconds until it is turned off. Include a way of turning the program on and off.

The answer has been hidden in the right side of the Patcher window. Scroll to the right or enlarge the Patcher window to see the answer.

Tutorial 7

Message Order



This lesson illustrates that messages in Max are always sent in *right-to-left* order. And, if a message triggers another object, that object will send *its* message(s) before anything else is done. Knowing these two principles can help you figure out exactly how a Patcher program is operating.

For example:

- Click on the **button** marked A. The bang message is first sent to the message box containing the number 60, that message is sent to the **print** object, and A: 60 is printed in the Max window. Then the bang message is sent to the message box containing the number 50, that message is sent to the **print** object, and A: 50 is printed in the Max window. Finally, the bang is sent to the message box containing the number 40, that message is sent to the **print** object, and A: 40 is printed in the Max window.

This illustrates the right-to-left order in which bangs are sent from the outlet of **button** to other objects, and also illustrates that the order of messages continues down the line until no more objects are triggered, (in this case, until the **print** object does its job), then goes back to the next patch cord coming out of the **button**, and the next bang is sent.

bangbang

The **bangbang** object sends a bang out *each* of its outlets when it receives any message. The number of outlets is specified by the typed-in argument. The order in which the messages are sent out the outlets is still right-to-left: the rightmost outlet sends first and the leftmost outlet sends last.

Tutorial 7

Right-to-left order

- Click on the **button** marked B, and you will see that when an object (such as **bangbang**) has more than one outlet, messages are sent out the outlets in right-to-left order.

When multiple patch cords are connected to a single outlet, as in examples A and C, messages are sent to the receiving objects in order of their right-to-left position, but when a single object has more than one outlet, as in examples B, D, and E, messages are sent out the outlets in right-to-left order, regardless of the destination.

trigger

The **trigger** object is very similar to **bangbang**, but deals with numbers as well as bangs. Instead of a single argument telling how many outlets there are, the number of outlets a **trigger** object has depends on how many arguments are typed in. Each argument in a **trigger** specifies what the output of an outlet will be: i for int, f for float, b for bang, or l for list (not shown in the example).

- Click on the message box 90, marked C. The **print** objects receive the number in right-to-left order, depending on their position.

- Click on the message box 90 marked D. Each outlet of the **trigger** has been assigned to send an int, so the number 90 will be sent out each outlet, in order from right-to-left.

- Click on the message box 90 marked E. In this example, each outlet has been assigned to send something different. The right outlet sends an int, the middle outlet sends a float, and the left outlet sends a bang.

Note: The names of **bangbang** and **trigger** can be shortened to **b** and **t** (as in example E). Max will still understand these object names.

Summary

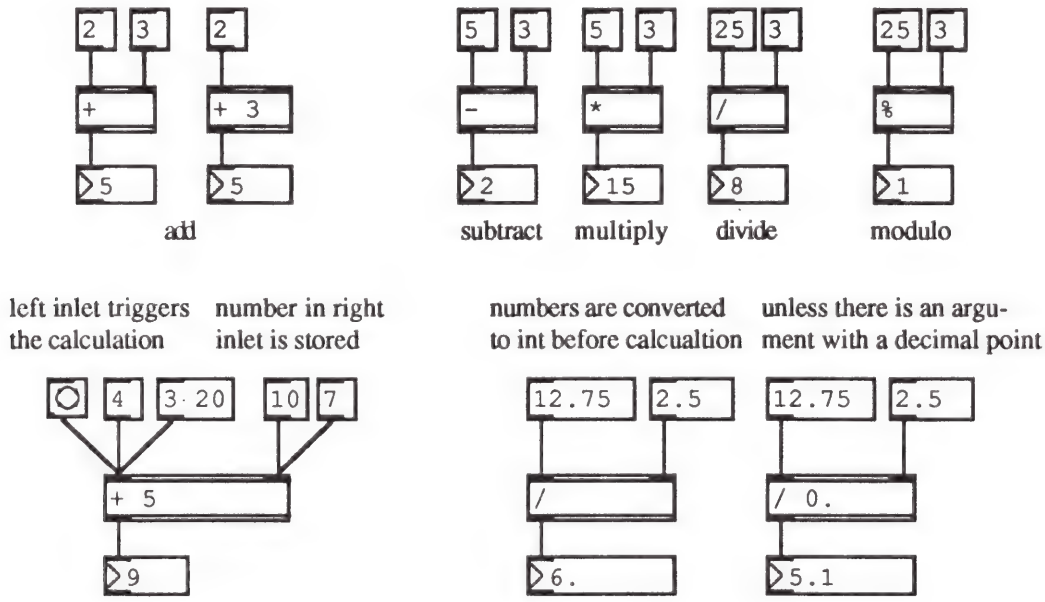
An object with multiple outlets sends messages out its outlets in order from *right-to-left*. When multiple patch cords are connected to a single outlet, the messages are sent in *right-to-left* order, depending on the position of the receiving objects. (If the receiving objects are perfectly aligned vertically, the order is *bottom-to-top*.) When the **bangbang** object receives any message, it sends a bang out each outlet. When **trigger** receives a number, a list, or a bang, it converts the message into the type assigned to each outlet before sending it out.

See Also

bangbang	Send a bang to many places, in order
buddy	Synchonize arriving numbers, output them together
fswap	Reverse the sequential order of two decimal numbers
swap	Reverse the sequential order of two numbers
trigger	Send a number to many places, in order

Tutorial 8

Arithmetic Operators



Max has an object for each of the basic arithmetic operations, plus a modulo operator (which gives the remainder when two integers are divided).

We call these objects *operators*—and the numbers they operate upon are called *operands*. Each operator object expects one operand as its argument and the other in its left inlet. The second operand can also be sent in through the right inlet. In the upper-left example, you see both methods.

Left Inlet Triggers the Object

Note that just connecting to an object's inlet does not perform any calculation. You have to *trigger* the calculation by sending a number (or bang) into the *left* inlet. The vast majority of objects are triggered by input received in the left inlet. Input received in the other inlets is usually stored for later use.

- In the upper examples, click on the message boxes above the operators.

Notice that the number coming in the right inlet has to be received *before* the number in the left inlet is received. That is because the message received in the left inlet triggers the calculation with the *most recently received* numbers. If you haven't supplied a number as a typed-in argument (and no number has been received in the right inlet), 0 is the default argument for the **+**, **-**, and ***** objects, and 1 is the default for **/** and **%**.

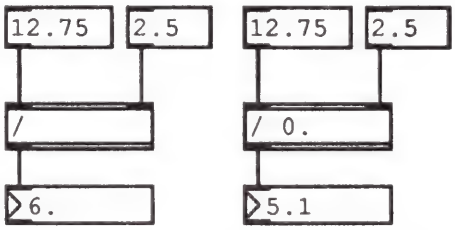
Tutorial 8

Doing math in Max

Int or Float Output

You may have noticed that the **/** object sends out 8 as the result of 25 + 3. That's because the output is an int, and is truncated before being sent out.

All the arithmetic operators send out an int as the result, unless they have a typed-in argument that contains a decimal point, in which case they are converted to float.



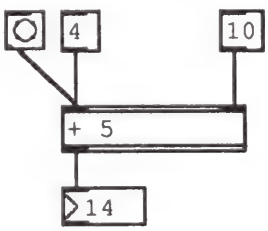
The two division programs at the bottom-right corner of the Patcher window demonstrate converting from one type to another. The first program removes the decimal part of any float numbers it receives. It performs the operation 12 + 2 and outputs a result of 6. The second program divides the numbers as floats and gives a full float output, because its typed-in argument contains a decimal point.

If you want an operator always to do float arithmetic operations, give the object an initial argument of a number with a decimal point, and then send the numbers you want it to use in through the left and right inlets.

bang **Message in Left Inlet**

The program in the bottom-left corner illustrates a couple of other features of operators.

- First, send the number 4 to the left inlet of the **+** object by clicking on the **message** containing 4. The object performs the caluclation 4 + 5 and outputs the result, 9.
- Next, send the number 10 to the right inlet. The number 5 is replaced by the number 10, but no output is sent. Only the left inlet triggers output.
- Now click on the **button** to send a bang to **+**. What happens? The bang causes **+** to "Do It!"—in this case, to do the calculation with the numbers it has most recently received.

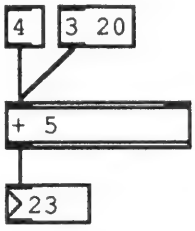


Tutorial 8

List in Left Inlet

Both operands can be sent to an operator *together*, as a list received in the left inlet. The operator will function exactly as if it had received the second number in the right inlet and the first number in the left inlet. The numbers are stored, the calculation is performed, and the result is sent out.

- Click on the message box containing 3 20 to see the effect of sending a list to the left inlet.



- Then send the number 4 to the left inlet, and you will see that the number 20 has been stored just as if it had been received in the right inlet.

This demonstrates that when you send a list of numbers to an object with more than one inlet, the numbers are distributed to the object's inlets from left to right. You will see other examples of this in future chapters.

Summary

Mathematical calculations are performed by *arithmetic operator* objects: **+**, **-**, *****, **/**, and **%**. The *operands* are received in the two inlets, but only the *left* inlet triggers output. A bang or a *list* in the left inlet can also trigger output. The operators send out an int, unless they have a float argument, in which case they send out a float.

Most objects in Max are triggered by input received in the *left* inlet. A *list* can be received in the left inlet, supplying values to more than one inlet at the same time.

Arithmetic operators are essential for any algorithm involving numerical calculation. Their use will be shown in future programs.

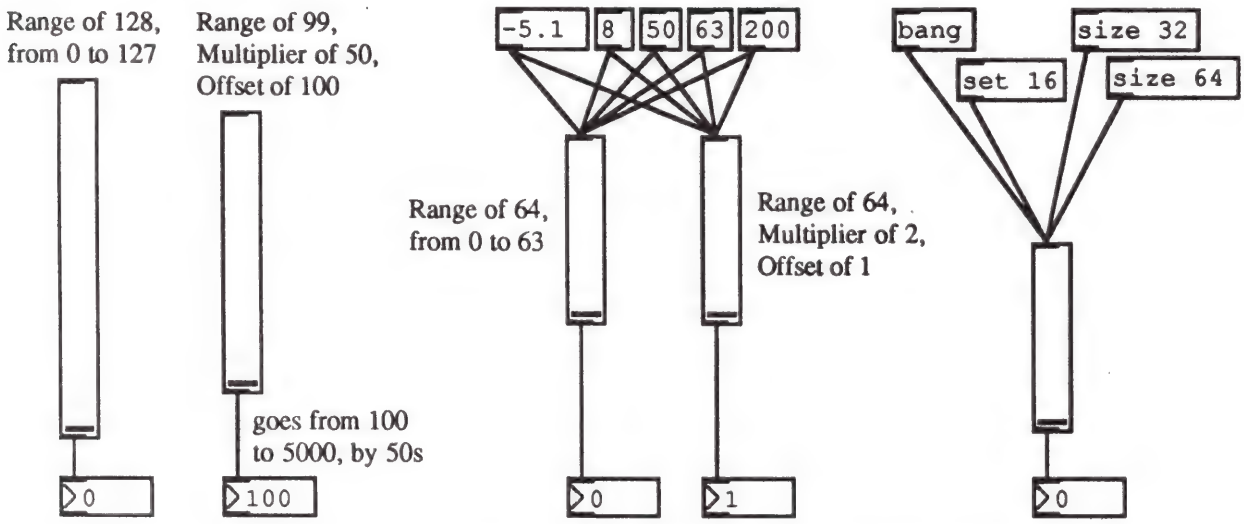
See Also

expr Evaluate a mathematical expression

Tutorial 9

Using the **slider**

Onscreen Controller



Clicking on the message box is one way of sending a number through a patch cord. Another object, the **slider**, lets you send any of a whole range of numbers by dragging with the mouse.

The **slider** object looks like this in the palette...

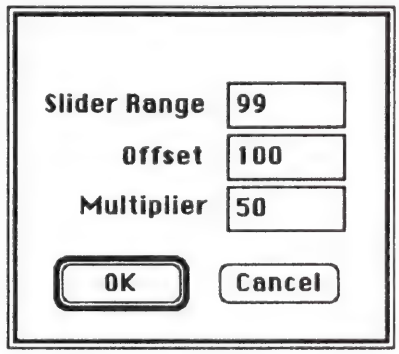
slider

When it is placed in the Patcher window it resembles a slider on a mixing console. Dragging on the **slider** sends out numbers as the mouse is moved.

- Click and drag on the first **slider** in the Patcher window, and see the output in the **Number box**.

When you create a new **slider**, its output ranges from 0 to 127. You can change the *Slider Range* by selecting the **slider** (when the Patcher window is unlocked) and choosing **Get Info...** from the Max menu. The **slider** automatically resizes itself to accomodate the specified range.

The **Get Info...** dialog box has two other values you can set: a *Multiplier*, by which all numbers will be multiplied before being sent out, and an *Offset*, which will be added to the number, after multiplication.



- The second **slider** in the Patcher window has a range of 99 (from 0 to 98), but before a number is sent out it is multiplied by 50, then has 100 added to it. So, when the slider is in the lowest position, it will output (0 * 50) + 100, which equals 100. When the slider is in the top position, it will output (98 * 50) + 100, which is 5000.

Many objects let you set options like this by with **Get Info...** dialogs.

Graphic Display of Numerical Values

In addition to responding to the mouse, the **slider** will move to whatever number it receives in its inlet. This makes it useful for graphically displaying the numbers passing through it. The *Multiplier* and the *Offset* are also applied to numbers received in the inlet, so the **slider** can also change values as they pass through.

- Click on the message boxes containing numbers, above the middle **slider** objects.

Notice that both **sliders** move to display the value they have received, but the numbers each sends is different. The **slider** on the left has an *Offset* of 0 and a *Multiplier* of 1, so it doesn't change the number it receives, but the other **slider** multiplies the incoming number by 2 and adds 1 to it.

Notice also that the numbers that are received and sent out can exceed the specified range of the **slider**, and that a float gets converted to int.

Other Inputs

A **slider** can receive other messages in its inlet. When it receives bang, it sends out whatever number it currently is displaying (with the *Multiplier* and *Offset* effects). The word set, followed by a number, sets the value of the **slider** without sending any output. The word size, followed by a number, changes the *Range* of the **slider** to that number.

Tutorial 9

Using the **slider**

Summary

A **slider** lets you output a continuous stream of numbers within a specified range by dragging on it with the mouse. It will also show and send out numbers received in its inlet, making it useful for graphically displaying the numbers passing through it.

By choosing **Get Info...** from the Max menu, you can change the *Slider Range*, and can also specify a *Multiplier*, by which all numbers will be multiplied before being sent out, and an *Offset*, which will be added to the number, after multiplication.

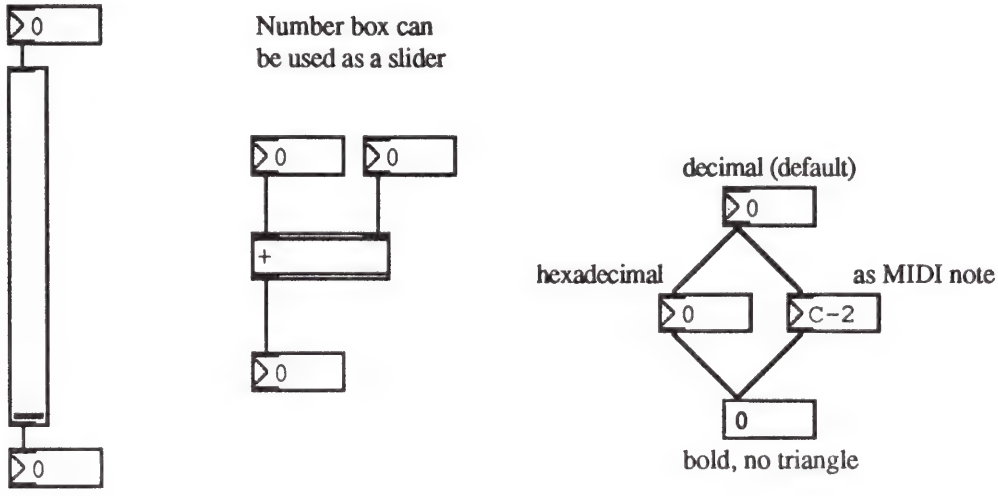
See Also

slider	Output numbers by moving a slider

Number boxes

Tutorial 10

Onscreen Controller



In the previous chapter we saw that a **slider** graphically displays the numbers passing through it, and can also send out numbers when you drag on it with the mouse. The **Number box** has these same capabilities.

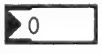
- Try dragging on the **Number box** at the top of the Patcher window, and you will see that it can be used as an onscreen controller much like the **slider**.

Unlike the **slider**, the **Number box** can have an unlimited range. You can produce any number with the **Number box** if you keep dragging.

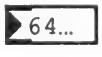
Type In Numbers

You can also type numbers into a **Number box** from the Macintosh keyboard.

- Click on the **Number box** at the top of the screen, without dragging. Notice that the triangle in the left edge of the **Number box** becomes black, showing that it has been selected.



- Type the number 64 on the Macintosh keyboard. The number will be followed by an ellipsis, indicating that the number has not yet been sent out the outlet.



- When you have finished typing in the number, you can send it out the outlet with any one of three actions: type the Return key, type the Enter key, or click anywhere in the Patcher window outside of the **Number box**.

- While a **Number box** is selected, you can also raise and lower the number in it by pressing the up and down arrow keys (↑ ↓). Holding down one of these arrow keys moves the number up or down continuously, just as if you were dragging on the **Number box** with the mouse.

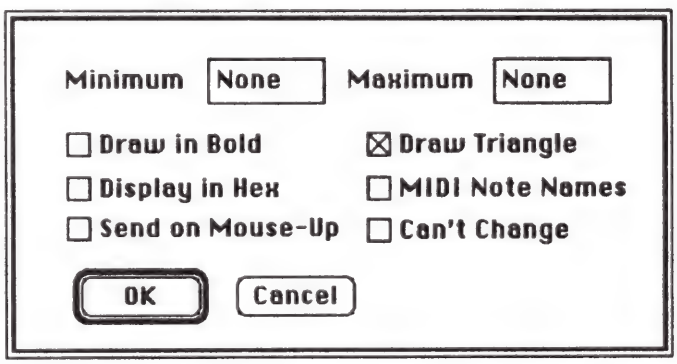
You can see that the **Number box** is useful both for displaying the numbers received in the inlet (as in the case of the **Number box** below the **slider**), and for allowing you to send numbers by typing them in or dragging with the mouse. The second patch shows the **Number box** in both uses—for sending numbers to the + object, and for displaying the result.

- Send a number to the right inlet of the + object, either by dragging on the **Number box** or by clicking on it and typing in a number. Remember, we want to send a number to the right inlet first, because the left inlet is the one that triggers the addition.

- Now send a number to the left inlet of the + object, and you will see the result of the addition in the bottom **Number box**.

Number box Range

You can set many characteristics of a **Number box**—how it functions and how it looks—by selecting a **Number box** and choosing **Get Info...** from the Max menu.



Get Info... dialog for a **Number box**

When you create a new **Number box**, it has an unlimited range. You can limit the range by typing a number into the *Minimum* and *Maximum* boxes in the **Get Info...** dialog box.

- Unlock the Patcher window, highlight the **Number box** located above the **slider**, then choose **Get Info...** from the Max menu.

- Type the number 0 into the *Minimum* box, and type the number 127 into the *Maximum* box. (You can move from one box to the other by typing the Tab key.) Click OK, then re-lock the Patcher window.

- Now when you drag on the **Number box**, it will not exceed the range of 0 to 127.

Tutorial 10

Note: The *Minimum* and *Maximum* settings of a **Number box** limit the range of numbers that can be sent out by dragging on it or by typing in a number. However, numbers *passing through* a **Number box** will still be passed through unchanged, even if they exceed the limits.

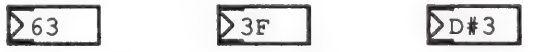
Display Options

The **Get Info...** dialog box has check boxes for toggling on and off various features. Some of the options affect the way the **Number box** functions, while others only affect the way it looks.

The *Draw Triangle* option is already checked, so that the triangle in the left edge of the **Number box** will make it visually distinct from the message box. Also, the triangle shows when a **Number box** has been clicked on, by turning black. The presence or absence of the triangle has no effect on the way the object functions, but it lets you change the appearance.

Draw in Bold displays the number in bold typeface. These aesthetic options can be used to emphasize certain **Number boxes**, or to show the user of your program which ones to drag on.

Display in Hex and *MIDI Note Names* cause the numbers to be displayed in a different format. (These options are available only in the int **Number box**.)



decimal (default) hexadecimal as MIDI note

Note: When a **Number box** is set to display in hexadecimal or MIDI note format, numbers entered by typing must be typed in that format.

Mouse Options

Normally the **Number box** sends out a continuous stream of numbers as it is being dragged upon with the mouse. The *Send on Mouse-Up* option causes the **Number box** to send out only the *last* number, the number that is showing when the mouse button is released. This lets you see the numbers as you drag, but only send out the single number that you choose.

When *Can't Change* is checked, numbers cannot be entered by dragging or typing. This is useful when you want a **Number box** to be for display only, without being an onscreen controller.

The third patch shows some of these options in use. The patch is for converting decimal numbers to their hexadecimal or note name equivalents, or vice versa.

- Drag on the top **Number box**, and you will see the numbers displayed in different formats.

Tutorial 10

Number boxes

Summary

The **Number box** can be used to display numbers passing through it, and/or as an onscreen controller for sending out numbers. Numbers can be sent out by dragging on the **Number box** with the mouse, or by clicking on the **Number box** and then typing in a number (or pressing the up or down arrow keys).

The range of numbers a **Number box** can send out can be specified by choosing **Get Info...** from the Max menu. With the **Get Info...** dialog you can also change how the numbers are displayed, and how the **Number box** responds to the mouse.

See Also

Number box	Display and output a number
Tutorial 3	About numbers

Test 2:
Temperature Conversion

Tutorial 11

Using Arithmetic Operators

To be sure you understand how to use arithmetic operators and the **Number box**, try this exercise:

1. Make a patch that converts a temperature expressed in degrees Fahrenheit into one expressed in degrees Celsius. Use a **Number box** to enter the Fahrenheit temperature, send the number to arithmetic operator objects to convert it, and use another **Number box** to display the result as a Celsius temperature.

Hints

The formula for converting Fahrenheit to Celsius is:

$$°C = (°F - 32) * 5/9$$

(The * is the multiplication operator.) You will first want to subtract 32 from the Fahrenheit temperature, then multiply the result by 5, then divide that result by 9.

Using Sliders

Here is a second exercise, a bit more difficult than the first one.

2. Make a patch that converts a temperature expressed in degrees Celsius into one expressed in degrees Fahrenheit. Limit the temperatures between the freezing point and the boiling point.

   In addition to using **Number boxes** to show the temperatures, use **sliders** as "thermometers" to show the temperatures graphically.

   Since the *Offset* and *Multiplier* features of the **sliders** can do addition and multiplication, try using these features to do some of the arithmetic work. Use as few arithmetic operator objects as possible.

Hints

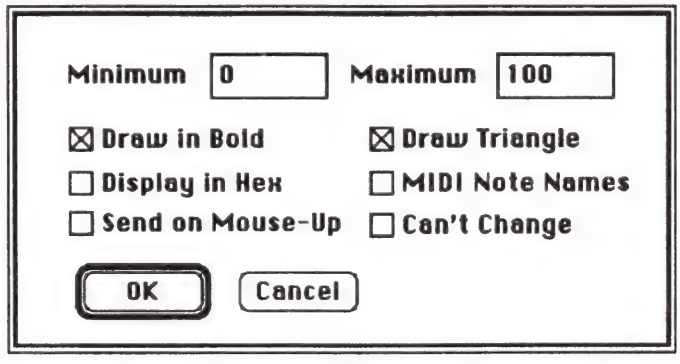
The formula for converting Celsius to Fahrenheit is:

$$°F = (°C * 9/5) + 32$$

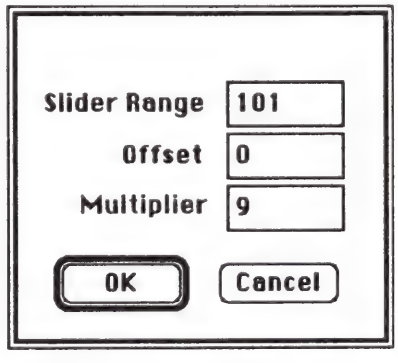
You will first want to multiply the Celsius temperature by 9, then divide the result by 5, then add 32 to that result.

In degrees Celsius, 0 is the freezing point and 100 is the boiling point. In degrees Fahrenheit, 32 is the freezing point and 212 is the boiling point.

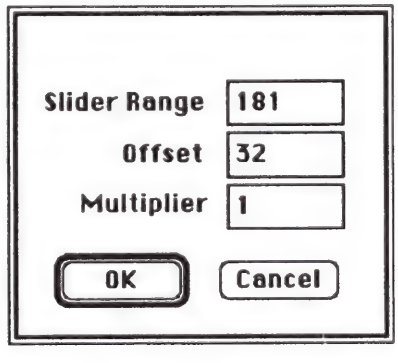
- Use the *Minimum* and *Maximum* features of the **Number box** to limit the input (Celsius temperature) between 0 and 100.



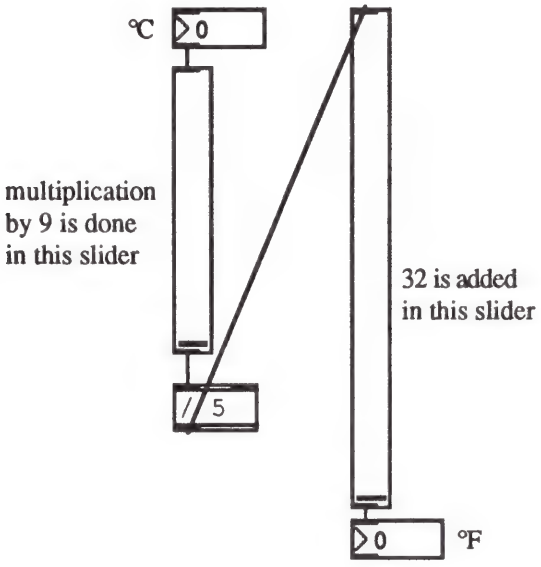
- Set the *Slider Range* of the **slider** which is depicting your Celsius "thermometer" to 101 so that it will display values from 0 to 100. (You can use the *Multiplier* feature of this **slider** to multiply the Celsius temperature by 9.)



- Use a **/** object to divide the Celsius temperature by 5. Then use the *Offset* feature of the **slider** which is depicting your Fahrenheit "thermometer" to add 32, and you will have the result. (Set the *Slider Range* to 181 so that it will range from 32 to 212.



Your objects will be connected something like this:



Scroll the Patcher window to the right to see solutions to these two exercises. Although temperature conversion is not a very useful musical function, these exercises exemplify how to solve a mathematical problem using operator objects.

In the subsequent chapters you will use these operators to manipulate MIDI data.

Summary

Arithmetic operators can be linked together to form a complete mathematical expression. The order in which the objects are linked is important for performing each operation in the proper order.

In some instances, the *Offset* and *Multiplier* features of the **slider** object can be used to perform an arithmetic operation.

Tutorial 12

Sending and receiving MIDI notes

Verify your MIDI Setup

Now that you've gotten a feel for how applications are constructed by connecting objects, we'll begin using MIDI data in our patches so that the examples have a more direct musical application.

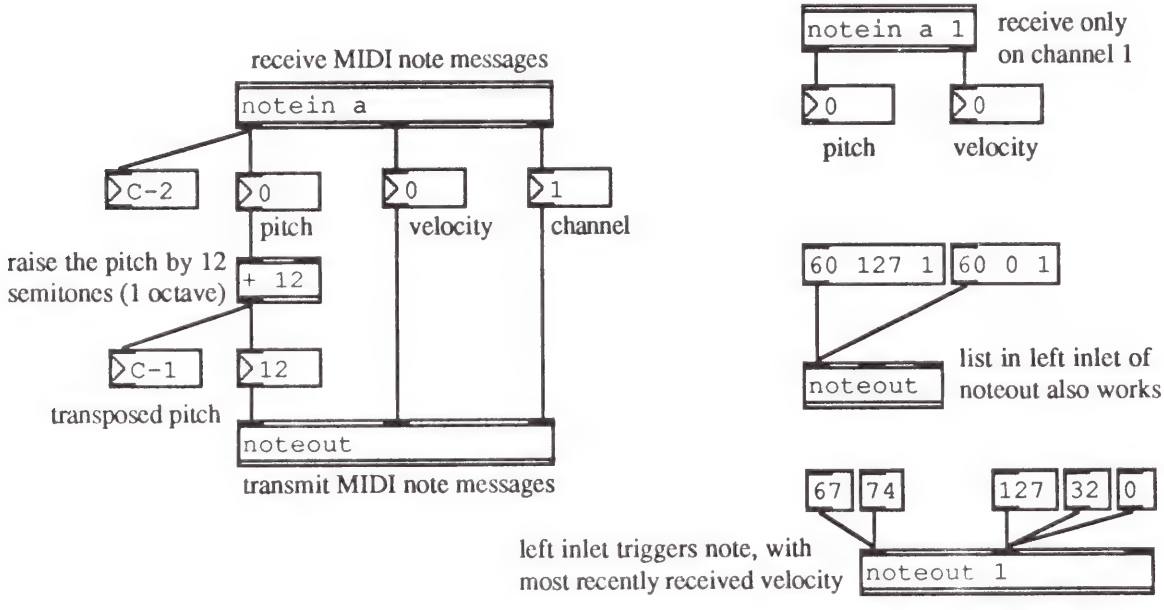
Make sure that your MIDI equipment is connected properly. If you have any doubts, review the beginning of *Basics*, entitled *Setting Up*, and review the first page of the Tutorial for a discussion of MIDI equipment.

MIDI Objects

There are many objects for transmitting and receiving data to and from your MIDI equipment. Objects which receive MIDI messages from your synth have no inlets, because their input comes from the serial ports or from the MIDI Manager, rather than from other Max objects. Likewise, objects which transmit MIDI messages have no outlets, since they send their messages out from Max.

The most basic MIDI objects are **midiin** and **midiout**, which receive and transmit raw MIDI data byte-by-byte, without analyzing the MIDI messages at all. More commonly, though, you will use more specialized MIDI objects, which filter the raw MIDI data coming into Max, and output only the the information you need.

For example, the **notein** object looks only for MIDI note messages, and when a note arrives, **notein** outputs the key number, the velocity, and the channel number. Similarly, the **bendin** object looks only for incoming pitch bend messages, and sends out the amount of pitch bend and the channel number.



Sending and receiving MIDI notes

Tutorial 12

notein and noteout

For the moment we will concern ourselves only with MIDI note data—receiving information about notes played on the synth, and transmitting messages to play notes on the synth.

- Play a few notes on your synth. You should see the note data in the **Number boxes**.

If you don't see anything happen, re-check your connections and your port assignments (choose **Midi Setup...** from the File menu) to make sure the MIDI data is being received by Max.

For the purpose of this Tutorial, your MIDI interface should be connected to the modem port of the computer. If you are using the MIDI Manager, port a of Max should be connected to the Modem port of the Apple MIDI Driver. If you are not using the MIDI Manager, port a of Max is automatically connected to the modem port.

The letter argument to **notein** indicates the port in which it receives MIDI note messages. If no argument is present, it receives from *all* ports.

In the title bar of a Patcher window, there is a MIDI Enable/Disable box that looks like a happy face or a MIDI plug.

When the box shows a MIDI plug icon, all MIDI objects are enabled and function normally. When the box looks like this...

...all MIDI receiving and transmitting objects in that window are disabled. Clicking on the box toggles it back an forth between Enabled and Disabled.

- Play a few notes and hold the keys down for a moment before releasing them. You can see that a message is sent for each note both when you press the key and when you release it. A note message with a velocity of 0 indicates a note-off.

The patch on the left is for playing in parallel octaves. Every note you play on the synth is received by the **notein** object. The pitch information is sent to a + object which adds 12 to the pitch value. This, of course, raises the pitch by 12 semitones (1 octave). We've included extra **Number boxes** so that you can see the pitch values both as numbers and as MIDI note names.

The velocity and channel information is passed on unchanged, and is reunited with the transposed pitch information in the **noteout** object. The new note, an octave higher than the one you played, and with the same velocity as you played, is sent back out to the synth by **noteout**.

Tutorial 12

Sending and receiving MIDI notes

Message Order

Although the pitch, velocity, and channel information appear to come out of **notein** at the same time, the numbers come out in right-to-left order (channel, then velocity, then pitch) just like any other object.

Like most objects, **noteout** is triggered by a message received in its left inlet—the pitch number. The pitch is combined with whatever velocity and channel values were most recently received in the other inlets, and a MIDI message is sent out to the synth.

This consistency of message ordering—outlets always send right-to-left, with objects triggered by the left inlet—allows different objects such as **notein** and **noteout** to communicate easily. Because the velocity and channel numbers come out of **notein** before the pitch does, they arrive at **noteout** before the pitch does, keeping all the data properly synchronized.

Receiving On One Channel

A **notein** object with no arguments, or with only a letter argument, receives incoming note messages on *all* channels. (This is known as *omni mode* in MIDI terminology.) You can set **notein** to receive on only one channel by typing in a channel number argument. When there is a channel argument, **notein** has only two outlets—for pitch and velocity—because the channel number is already known. Both the port argument and the channel argument are optional.

- If your MIDI keyboard can transmit on different channels, set it to transmit on some channel other than 1. Now when you play notes the **notein** on the left still receives them, but the **notein** on the right ignores them.

Transmitting Note Messages

You don't necessarily need to play notes into Max to send notes out. You can transmit notes to the synth which are produced within Max.

One way to do this is to send a list—consisting of pitch, velocity, and channel—to the left inlet of **noteout**. You may remember this use of lists with the arithmetic operator objects in *Tutorial 8*.

- Click on the message boxes containing lists. One list sends a note-on, and the other sends a note-off (a note with a velocity of 0). It is necessary (or at least neighborly) to follow a note-on with a note-off, otherwise the note will continue to play. Try this with a sustained sound on your synth.

The last patch demonstrates that you can type in an argument for the channel on which **noteout** will transmit. The channel inlet is still present, however, and you can change the channel by sending in a new number.

The patch also shows that **noteout** combines pitches with whatever velocity was most recently received.

- Try sending different velocities to **noteout**. The velocity is just stored until a pitch number is received to trigger a MIDI note message.

Summary

The **notein** object looks for incoming MIDI note messages, and outputs pitch, velocity, and channel data. You can type in a specific port letter as an argument, which causes **notein** to output only the note data received in that one port. You can also type in a specific channel number, causing **notein** to output only the note data received on that one channel.

When the **noteout** object receives a number in its left inlet, it uses that number as a pitch value, combines the pitch with a velocity and a channel number, and transmits a MIDI note message. The pitch, velocity, and channel can also be received together as a list in the left inlet.

See Also

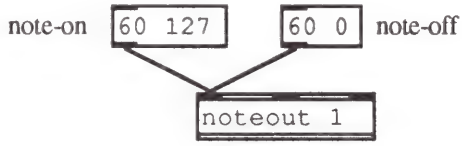
notein	Output incoming MIDI note messages
noteout	Transmit MIDI note messages

Tutorial 13

Managing note data

Note-On and Note-Off Messages

One of the main problems that you encounters when sending note messages to the synth from Max is the need to follow every note-on message with a corresponding note-off message. For example, just sending a pitch and a velocity to **noteout** plays a note on the synth, but that note will not be turned off until you also send the same pitch with a velocity of 0.

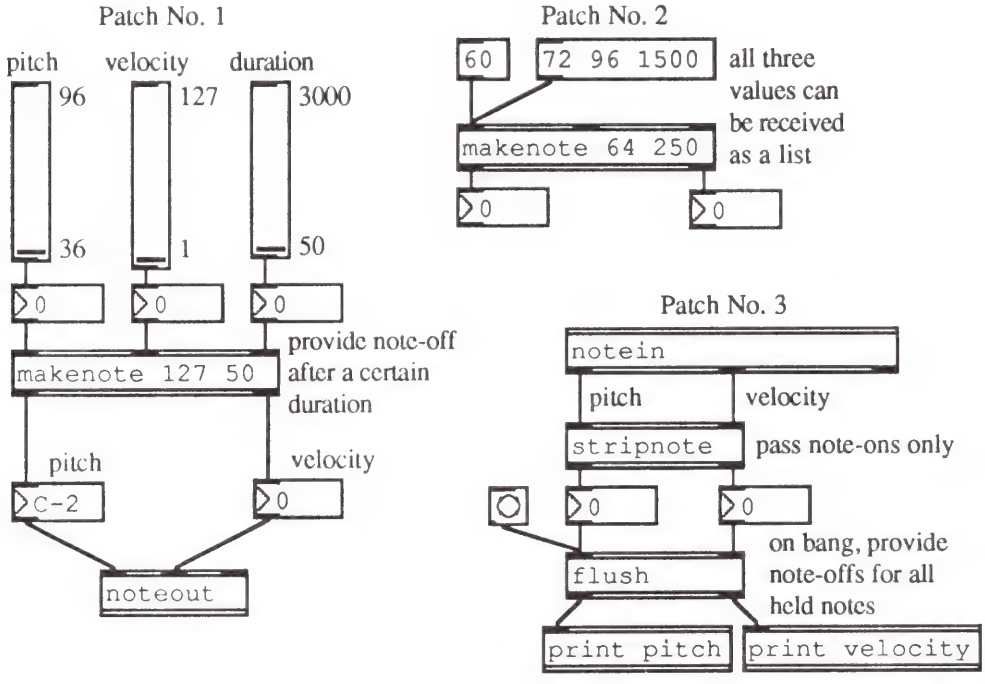


makenote

Max has objects which generate note-off messages for turning off notes which have been sent to the synth. One such object is **makenote**.

When **makenote** receives a number in its left inlet, it uses the number as a pitch value, combines that pitch with a velocity, and sends the numbers out its two outlets. Then after a specified delay (or *duration*), **makenote** automatically sends the same pitch number, but with a velocity value of 0.

The synth interprets a note-on message with a velocity of 0 as a note-off. So, when the output of **makenote** is sent to **noteout**, both a note-on and a note-off get transmitted to the synth.



- Drag on the **slider** marked pitch in Patch 1. Each number that comes out of the **slider** is combined with a velocity by **makenote** (in this case, the velocity is 127, specified in the first argument), and the pitch and velocity are sent to **noteout**.

  50 milliseconds after each pitch is received (the duration specified in the second argument) **makenote** sends the same pitch out again, with a velocity of 0. The result is that every note has a duration of 50ms.

The velocity and the duration can be changed by numbers received in the middle and right inlets. The most recent values received in these inlets are used the next time a pitch is received in the left inlet.

- Try changing the velocity and duration by dragging on the **sliders,** then play more notes by dragging on the pitch **slider**. The notes now have the velocity and duration you specified.

Note: When no channel number has been specifed to **noteout**, either as a typed-in argument or in the right inlet, it is set to channel 1 by default.

Patch 2 demonstrates that the pitch, velocity, and duration values can all be received in the left inlet as a list.

- Click on the message box containing the number 60. You can see that it is combined with a velocity of 64, then combined with a velocity of 0 after 250 milliseconds.

- Click on the message box containing the list. The pitch 72 is sent out with a velocity of 96, and after 1.5 seconds it is sent out again with a velocity of 0.

- Now click again on the number 60. You can see that the velocity and duration values (96 and 1500) have been stored in **makenote**, and are applied the pitch received in the left inlet.

stripnote

The **stripnote** object is sort of like **makenote** in reverse. It receives a pitch and a velocity in its inlet, and passes them on only if the velocity is not 0. In this way, it filters out note-off messages, and passes only note-on messages.

This is useful if you want to get data only when a key on your keyboard is pressed down, but not when the key is released. For example, you might want to use a pitch value from the keyboard to send a number to some object in Max, but you wouldn't want to receive the number both from the key being pressed *and* from the key being released.

flush

The **flush** object is another object for generating note-off messages. Unlike **makenote**, however, it does not generate them automatically after a certain duration. Instead, **flush** keeps track of the notes that have passed through it. When it receives a bang in its left inlet it provides note-offs for any notes that have not yet been turned off.

Tutorial 13

Managing note data

Both **flush** and **stripnote** receive velocity values in the right inlet and pitch values in the left inlet, and pass the same type of values out the outlets. They are triggered by a pitch value received in the left inlet, and use the velocity value that was most recently received in the right inlet. Both objects can also receive the pitch and velocity values *together* as a list in the left inlet.

- Play a few notes on your MIDI keyboard. You can see that **stripnote** passes only the note-on messages and suppresses the note-offs. The note-ons get passed through **flush**, and are received by the **print** objects. (A **flush** object will also pass on any note-offs it receives, but in this case **stripnote** has filtered them out.)

- Now click on the **button** to send a bang to the left inlet of **flush**. The **flush** object keeps track of all the note-ons it has received which have not been followed by note-offs, and when a bang is received, **flush** provides note-offs for those held notes.

The advantage of sending pitch and velocity pairs through **flush** before sending them to **noteout** is that **flush** has no noticeable effect until it receives a bang, then it turns off any notes that are still on. This is useful for avoiding stuck notes.

Summary

A MIDI note-on message transmitted by **noteout** should be followed by a corresponding note-off message, so that the note played on the synthesizer gets turned off.

The **makenote** object combines pitch values with velocity values, to be sent to **noteout**. After a certain duration, the same pitch is sent with a velocity of 0, to turn off the note. The **stripnote** object is the opposite of **makenote**. It filters out note-off messages (pitch-velocity pairs in which the velocity is 0), and passes on only note-on messages (messages with a non-zero velocity).

The **flush** object keeps track of the notes that have passed through it, and when it receives a bang it sends out a note-off for any notes which are still on.

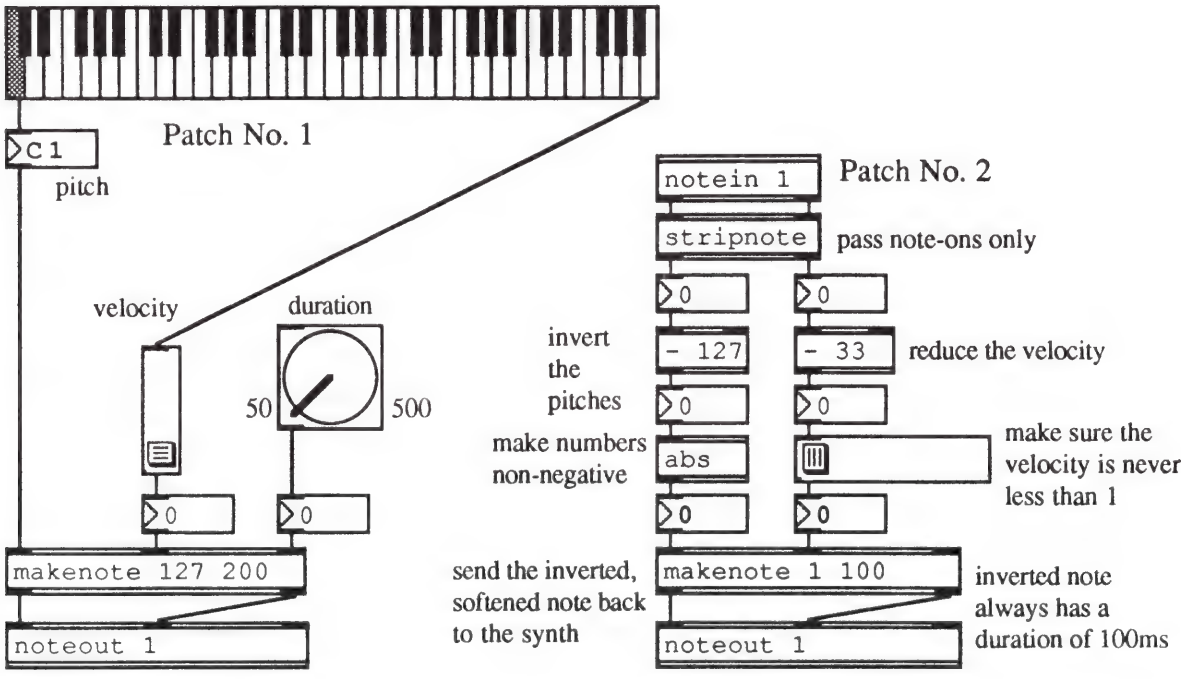
See Also

flush	Provide note-offs for held notes
makenote	Generate a note-off message following each note-on
poly	Allocate notes to different voices
stripnote	Filter out note-off messages, pass only note-on messages
sustain	Hold note-off messages, output them on command

Tutorial 14

Diverse Onscreen Controllers

In this tutorial, we'll introduce some objects that function similarly to the **slider**, but differ somewhat in appearance and behavior.



kslider

Patch No. 1 is similar to the patch in the previous chapter. It allows you to play notes with the mouse. However, this patch uses a keyboard slider, **kslider**.

- Try playing notes by clicking and/or dragging on the **kslider**. It has been set to output numbers from 36 to 96 (MIDI notes C1 to C6) out its left outlet. The numbers are then sent to the left inlet of **makenote**, where they are paired with a velocity (from the right outlet of **kslider**), and the notes are sent to **noteout**.

- When you drag along the lower half of **kslider**, it outputs only the the numbers associated with the white keys. When you drag along the upper half, it plays both white and black keys.

- The velocity that is sent out the right outlet depends on how high the mouse is placed on the key you are playing.

The *Range* and *Offset* of the notes displayed by **kslider** can be changed by choosing **Get Info...** from the Max menu. The *Offset* is the value that will be output by clicking on the lowest note of the **kslider**, and is specified as a MIDI note name. The default is C1 (36). If you want an offset of 0, set it to C-2. The *Range* is specified as the number of octaves you want the **kslider** to have. The **Get Info...** dialog also lets you select one of two sizes for kslider, *Large* or *Small*.

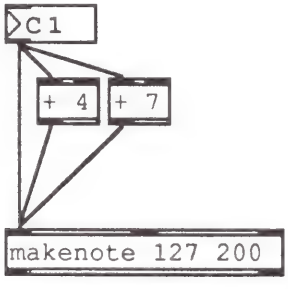
Range and *Offset* refer only to the numbers displayed by **kslider**, or sent out its outlet by clicking and dragging with the mouse. Numbers received in the inlet of **kslider** are unaffected by the *Offset*, and are passed through unchanged.

Playing Parallel Chords

Suppose you wanted **kslider** to play parallel major triads. How would you go about it?

In addition to sending the numbers directly to the left inlet of **makenote**, you can also send them to two different + objects. One + object can add 4 to the number (raising the pitch a major third), and the other can add 7 (raising the pitch a perfect fifth). These transposed pitches are then sent to **makenote**, along with the original pitch.

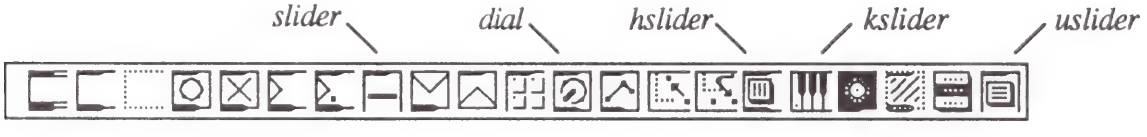
- Try it yourself. Unlock the Patcher window and create two new + objects just above **makenote**. Then connect the outlet of the **Number box** to the inlets of the + objects, and connect the outlets of the + objects to the left inlet of **makenote**.



- Now lock the Patcher window and click on a note of the **kslider** to hear the results. You can also try changing the numbers you add with the + objects, to create other types of triads.

dial, hslider, and uslider

In Patch 1 the velocity values are displayed by a slider object named **uslider**, and the durations are supplied by a **dial**. Patch 2 contains the horizontal slider, **hslider**.



Various sliders and dials in the palette

There are a few important differences between these objects and the **slider** and **Number box** objects seen in previous chapters.

1. The **slider** and the **Number box** send out numbers when you drag them with the mouse. You can drag on the other sliders and dials, but you can also change them with a single mouse click.

Tutorial 14

- Click on the sliders and dials, and notice how they jump to the new position and send out a number, even without dragging the mouse.

2. The **slider** and **kslider** objects resize themselves automatically depending on their range. The **hslider** and **uslider** can be shrunk or enlarged to virtually any size with the grow bar, regardless of the range of numbers they send out. The **dial** has only one possible size, regardless of its range.

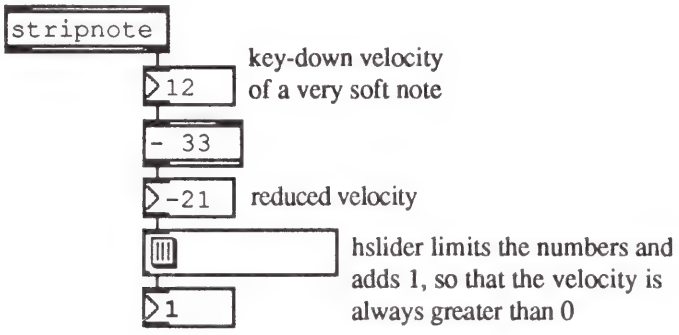
3. Although the **slider**, **kslider**, and **Number box** may have a limited range of numbers that can be output by dragging, they do not limit the range of numbers that can *pass through* them. The **dial**, **hslider**, and **uslider** *do* limit the numbers received in their inlets. Any incoming number that is less than 0, or that exceeds the specified range, will be automatically restricted within those limits.

The limiting feature can be put to use, as is shown in Patch 2. Let's analyze what the patch does.

Analyzing Patch 2

- Play a scale on your MIDI keyboard. Notice that as you play you also hear a scale of short notes moving in the opposite direction.

When you play notes on the synth, the pitch and velocity are sent through **stripnote**, which filters out all the note-off messages, passing only the note-ons. Then 33 is subtracted from the velocity.



- Play some notes very, very softly so that your key-down velocity is less than 33.

This results in negative numbers coming out of the - (minus) object. The **hslider** limits the numbers it receives in its inlet, so that none of them is less than 0, and **hslider**'s *Offset* of 1 ensures that all velocities are at least 1. The reduced velocity finally arrives in the middle inlet of **makenote** and is stored there.

Next, the pitch value comes out of **stripnote**, and has 127 subtracted from it. This means that pitches, which usually range from 0 up to 127, will range from -127 up to 0. If you have a 61-note keyboard, your pitches range from 36 up to 96, and subtracting 127 from them causes them to range from -91 up to -31.

This number is then sent to an **abs** object, which sends out the absolute (non-negative) value of whatever number it receives. So now, instead of your pitches ranging from -91 up to -31, they range from 91 *down* to 31. As you play higher on the keyboard, the numbers being sent to **makenote** become lower, and vice versa.

The inverted pitches are paired with the reduced velocity in **makenote**, and the notes are sent out, then are turned off after 100 milliseconds ($^1/_{10}$ of a second).

Summary

The **hslider** and **uslider** objects are similar to slider, but can be made any size. **kslider** is a keyboard-like slider, the *Range* of which is specified as a number of octaves. You can perform both chromatic glissandi and diatonic glissandi (white-keys only) on **kslider**.

The **dial**, **hslider**, and **uslider** objects all limit the numbers they receive in their inlet. Numbers which exceed the range of these sliders are set to the minimum or maximum value of the slider. Unlike **slider**, these other sliders respond to a single mouse click, without dragging.

The **abs** object sends out the absolute value of whatever number it receives in its inlet. The limiting sliders and **abs** represent two different ways to avoid negative numbers. (Other objects which can serve this purpose are **maximum** and **split**.)

See Also

abs	Output the absolute value of the input
dial	Output numbers by moving a dial onscreen
hslider	Output numbers by moving a slider onscreen
kslider	Output numbers from a keyboard onscreen
split	Look for a range of numbers
uslider	Output numbers by moving a slider onscreen

Making decisions with comparisons

Tutorial 15

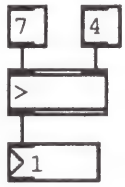
Relational Operators

One of the most basic things a computer program does is perform some kind of a test, then make a decision based on the result of that test. The test is usually some kind of *comparison*, such as seeing if two numbers are equal. The answer to this test can be used to determine what the computer does next.

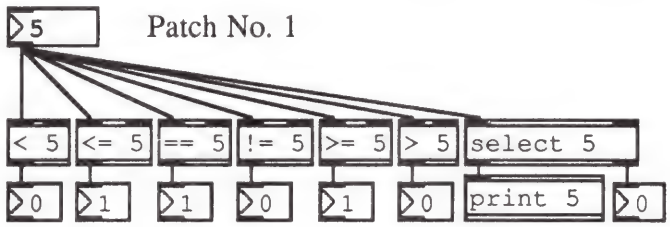
Numbers are compared using *relational operators* which characterize the relationship of one number to another with such terms as *is less than*, *is greater than*, *is equal to*, etc. Max has several relational operator objects, for comparing one number to another:

< means *is less than*	<= means *is less than or equal to*	== means *is equal to*
> means *is greater than*	>= means *is greater than or equal to*	!= means *is not equal to*

Max's relational operator objects send out the number 1 if the statement is true, and 0 if the statement is false. So, for example, to test the statement *7 is greater than 4*, you would send the number 4 to the right inlet of a > object, then trigger the object by sending the number 7 in the left inlet. Since the statement *7 is greater than 4* is true, the objects sends out the number 1.



The right operand can also be provided as an argument typed into the object box.



- Drag on the **Number box** at the top of Patch 1. Notice especially the output of each object as you pass by the number 5.

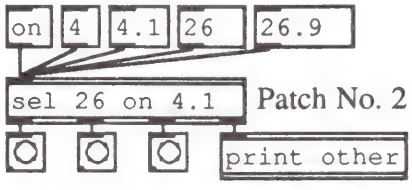
The relational operators normally expect to receive ints in the inlets. Floats are converted to int before the comparison is made. Like the arithmetic operators, however, the relational operators can compare floats if there is a float argument typed in.

select

The **select** object is a special relational operator. If the the left operand is equal to the right operand, a bang is sent out the left outlet. Otherwise, the left operand is passed out the right outlet. The effect is that every number received in the left inlet gets passed on out the right outlet *except*

the one **select** is looking for. When **select** receives the number it's looking for, it sends a bang out the left outlet.

Patch 2 shows that the **select** object (whose name can be shortened to **sel**) can actually be given *several* arguments (as many as 10), and each argument can be an int, a float or a *symbol* (i.e., a word). The input is converted to the proper type (int, float, or symbol) before being compared to each argument. Notice that the right inlet is not present if there is more than one argument.



Patch No. 2

- Click on the different messages. Notice that if the input matches one of the arguments, a bang is sent out the outlet that corresponds to that argument. If there is no match, the input is passed out the right outlet.

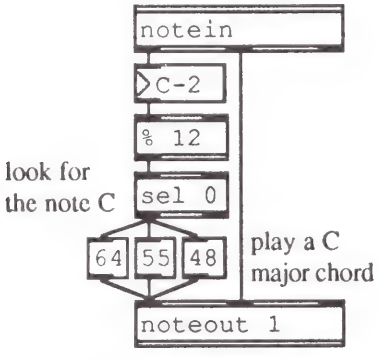
When the input is an int (such as 4) it is converted to float before being compared with a float argument. A float input (such as 26.9) is truncated before being compared to an int argument.

Using Comparisons to Make Decisions

The **select** object sends out a bang, which can be used to trigger other objects, and relational operators send out the numbers 1 and 0, which can be used to toggle something on and off (such as a **metro**). So you can see that comparisons can be used in a patch to decide when to trigger another object.

Patch 3 shows the use of **sel** to look for a certain pitch being played on your MIDI keyboard.

Patch No. 3



The pitch is first sent to a **%** operator, which divides it by 12 and sends out the remainder. Since the note C always has a MIDI key number which is a multiple of 12 (such as 36, 48, 60, etc.), the output of the **% 12** object will be 0 whenever the note C is played.

Tutorial 15

Each time **sel** receives the number 0 from **%**, it sends a bang to the message boxes, which send the notes C2, G2, and E3 (48, 55, and 64) to **noteout**. These pitches are combined with the velocity of the note C that is being played on the synth, so the chord has the same velocity and duration as the note being played.

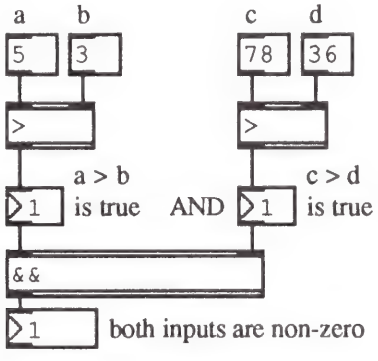
In this example, we test to see if the pitch being played is equal to C. When this is true, the chord is triggered.

Combining Comparisons with "Or" or "And"

The object **||** means *or*. If either the left operator *or* the right operator is non-zero (*true*), **||** sends out the number 1. If both operators are 0, it sends out 0.

The object **&&** means *and*. If the left operator *and* the right operator are both non-zero, **&&** sends out the number 1. Otherwise, it sends out 0.

|| and **&&** are used to combine two comparisons into a single statement, such as: *a is greater than b AND c is greater than d.*



Note that in the example above, the number 5 (*a*) must be sent *last*, so that all the other values will have arrived when **&&** is triggered.

Patch 4 is similar to Patch 3, but it uses **||** to look for two pitches instead of one. The patch says, *if the pitch played is B OR it is D, then play the notes G1 and F3*. The effect, of course, is to accompany the notes B and D with an incomplete dominant seventh chord.

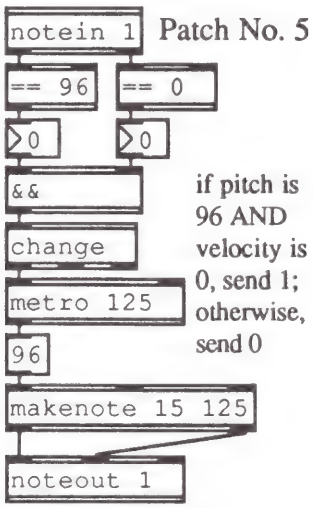
- Play a melody in the key of C on your synth. Patches 3 and 4 provide you with an annoyingly Haydnesque accompaniment.

Using Comparisons to Toggle an Object On and Off

Patches 3 and 4 demonstrate that any number—and thus any key or combination of keys on the synth—can be used to trigger something in Max. Similarly, the 1 and 0 sent out by relational operator objects can be used to turn an object such as **metro** on and off. Patch 5 demonstrates this idea.

Tutorial 15

Making decisions with comparisons



- Play the note C6 (high C) on the synth. As soon as you release the key, Patch 5 begins playing the note repeatedly until the next time you play a note.

The patch is looking for the condition when the pitch is equal to 96 *and* the velocity is equal to 0. When both conditions are true, **&&** sends out 1, otherwise **&&** sends out 0. Obviously, the vast majority of note messages will cause **&&** to send out 0. In order to avoid sending the number 0 to **metro** over and over unnecessarily, the output of **&&** is first sent to a **change** object. The purpose of **change** is to filter out repetitions of a number. The number received in the inlet is sent out the left outlet only if it is different from the preceding number.

When **metro** is turned on it sends the number 96 to **makenote** at the rate of 8 notes per second (once every 125 ms).

Summary

Relational operators—**<**, **<=**, **==**, **!=**, **>=**, and **>**—compare two numbers, and report the result of the comparison by outputting 1 or 0. The **&&** and **||** objects test whether their inputs are 0 or non-zero, making them useful for combining two comparisons into a single test.

The **select** object (also known as **sel**) looks for certain numbers (or *symbols*). If the input matches what it is looking for, it sends a bang out one of its outlets. Otherwise, it passes the input out its right outlet.

The results of any of these comparisons can be used by the program to *make a decision* whether to trigger other objects.

The **change** objects passes on a number received in its inlet, only if the number is different from the preceding one.

Tutorial 15

See Also

change	Filter out repetitions of a number
select	Select certain inputs, pass the rest on
<	*Is less than*, comparison of two numbers
<=	*Is less than or equal to*, comparison of two numbers
==	Compare two numbers, output 1 if they are equal
!=	Compare two numbers, output 1 if they are not equal
>=	*Is greater than or equal to*, comparison of two numbers
>	*Is greater than*, comparison of two numbers

Tutorial 16

More MIDI ins and outs

Introduction

There are many MIDI objects besides **notein** and **noteout**. Objects exist for receiving and transmitting any kind of MIDI message. In this chapter, we introduce a few of these objects: **bendin** and **bendout** for pitchbend messages, **pgmin** and **pgmout** for program change messages, and **ctlin** and **ctlout** for continuous controller messages.

Like **notein** and **noteout**, these other objects can be given optional arguments to specify the port and MIDI channel on which they will operate. When a channel number is specified as an argument in a MIDI receiving object, the outlet for sending the channel number disappears.

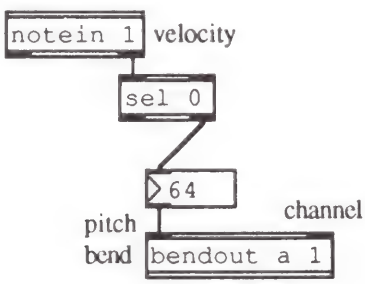
channel number comes out right outlet when no channel number is specified

right outlet is unnecessary when a channel number is specified

bendin and bendout

bendin receives data from the pitch bend wheel of your MIDI keyboard. The channel is sent out the right outlet, and the pitch bend data (the amount of pitch bend) is sent out the left outlet. Pitch bend data ranges from 0 to 127, with 64 meaning no bend at all.

The first patch demonstrates how easily one kind of MIDI data can be given a different meaning. In this case, the velocity of the notes played on the synth is sent to **bendout** to control the pitch bend.



- Play a single note repeatedly on the synth. The pitch is bent upward when you play hard (when the velocity is greater than 64), and is bent downward when you play softly.

Notice the way that **sel** is used to filter out note-off velocities. If this were not done, the pitch would be bent down to 0 each time a key is released, which might be bothersome in some cases. (On the other hand, triggering pitch bends with note-offs could be an interesting effect.)

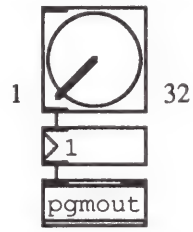
pgmin and pgmout

MIDI program change messages change the sound a synthesizer uses to play notes. Almost all synths can receive program change messages, and many can send them as well.

Tutorial 16

Different synths have different numbers of possible sounds, and have different ways of numbering their sounds. Some synths start numbering sounds from 0, while others start from 1. Others use unique numbering systems, such as a letter-number combination simulating base-8 arithmetic, etc.

- Use the **dial** in the second patch to send program change messages to the synth.

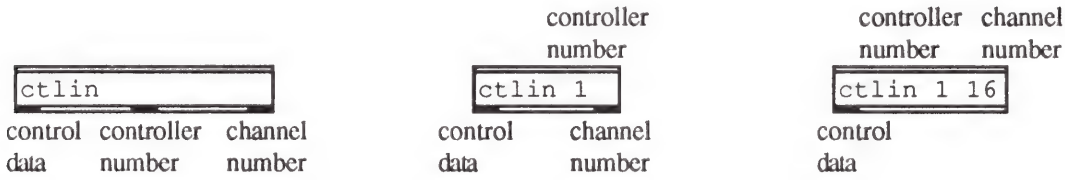


- The **dial** sends out numbers from 1 to 32. If this is not appropriate for your synth, you can change the *Dial Range* and *Offset*. Unlock the Patcher window, select the **dial**, and choose **Get Info...** from the Max menu.

ctlin and ctlout

A control change message contains three vital items of information: the channel, the controller number, and the control data. The meaning of the data is dependent on the controller number. For example, controller number 1 is usually assigned to the modulation wheel, controller 7 to volume, etc.

Therefore, in addition to port and channel arguments, **ctlin** and **ctlout** can be given a specific controller number as their first argument. When a specific controller number is given as an argument to **ctlin**, the controller number outlet disappears. For more about the arguments and their default values, look under **ctlin** and **ctlout** in *Objects*.

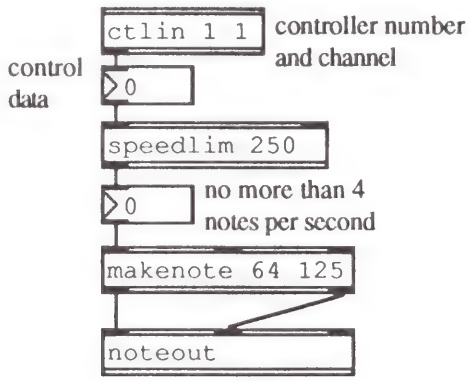


Tutorial 16

More MIDI ins and outs

Reassigning Control Data

You can use a continuous MIDI controller to send a stream of numbers to Max, then use those numbers in any way you like. In this patch, the data from the mod wheel of the synth is used to send pitch values back to the synth.



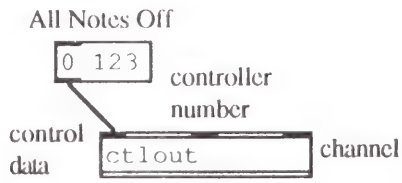
- Move the modulation wheel on your MIDI keyboard, and you should hear notes play.

The **speedlim** object limits the speed with which numbers can pass through it. When **speedlim** receives a number, it sends the number out the outlet, then waits a certain number of milliseconds before it will receive another number. The number of milliseconds between numbers can be a typed-in argument and/or supplied in the right inlet.

Channel Mode Messages

Controller numbers 122 to 127 are reserved for special MIDI commands known as *channel mode messages*. Channel mode messages can be received and transmitted with **ctlin** and **ctlout**, just like any other control message.

The last patch shows an example of **ctlout** used to transmit a channel mode message meaning *All Notes Off* (controller number 123 with a value of 0). Many synths (but not all) recognize this message and turn off all notes currently being played. For turning off notes within Max, it's more reliable to use an object such as **flush** or **poly**.



The patch also demonstrates that **ctlout** (and the other transmitting objects) can receive values for all inlets in the form of a list in the left inlet. When there are no arguments, **ctlout** transmits out port a on channel 1.

Summary

Pitch bend messages are received and transmitted with **bendin** and **bendout**, program changes with **pgmin** and **pgmout**, and continuous control messages (and *channel mode messages*)with **ctlin** and **ctlout**.

MIDI data can be altered and reassigned in any way within Max.

A stream of numbers can be "slowed down" by filtering them with **speedlim**, which ignores some of the numbers if they arrive too fast. This is a good method of converting a continuous stream of numbers into regular, discrete steps.

See Also

bendin	Output incoming MIDI pitch bend values
bendout	Transmit MIDI pitch bend messages
ctlin	Output incoming MIDI control values
ctlout	Transmit MIDI control messages
pgmin	Output incoming MIDI program change values
pgmout	Transmit MIDI program change messages
speedlim	Limit the speed with which numbers can pass through

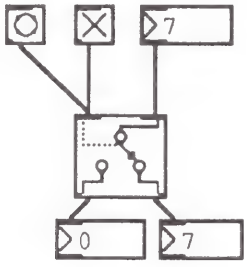
Tutorial 17

Gates and switches

Ggate

In Tutorial 15 we saw examples of how to use comparisons to make a decision whether to send a message. It is also possible for objects to make decisions about *where* to send a message.

The patch in the upper-left corner shows a graphical object, **Ggate**, for routing incoming messages out one outlet or the other. Messages received in the right inlet are sent out whichever outlet is pointed at by the arrow. The direction of the arrow can be changed by clicking on **Ggate** with the mouse, sending a bang to the left inlet, or sending a zero or non-zero number to the left inlet.

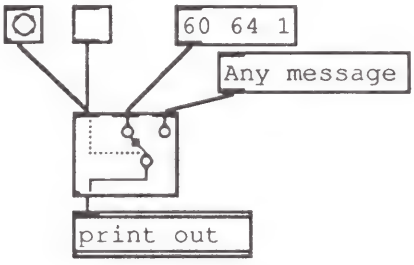


When the number in the left inlet is 0, the arrow points to the left outlet. A non-zero number in the left inlet causes the arrow to point to the right outlet. In the example above, the **toggle** has sent the number 1 to the left inlet of **Ggate**, causing the arrow to point to the right outlet. Consequently, any message received in the right inlet is passed out the right outlet.

- Try the various methods of changing the direction of the arrow, then drag on the **Number box** to send numbers to the right inlet of **Ggate**.

Gswitch

The second patch shows a similar object, **Gswitch**, which can open one of two *inlets*. Whichever inlet is being pointed to by the arrow is the *open* inlet, and messages received in that inlet are passed out the outlet. Messages received in the closed inlet are ignored. The leftmost inlet is the *control inlet*, for switching the arrow back and forth. It functions like the left inlet of **Ggate**.



- Click on the two message boxes to send messages to **Gswitch**. Only the message received in the open inlet is sent out the outlet. Change the direction of the arrow and try again.

Ggate and **Gswitch** will pass on any type of message—numbers, lists, and text.

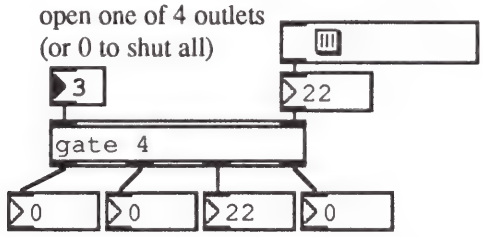
Tutorial 17

gate

The **gate** object is like **Ggate**, with a few important differences:

1. The number of outlets is determined by the argument to **gate**. (There can be as many as 10 outlets.) A single outlet is opened when its number is received in the left inlet. All other outlets are closed.
2. When the number 0 is received in the left inlet, *all* outlets are closed.
3. A **gate** does not respond to a mouse click the way **Ggate** does.

- Send a number to the left inlet of gate to open one of the outlets (or 0 to close all outlets), then send numbers to the right inlet.

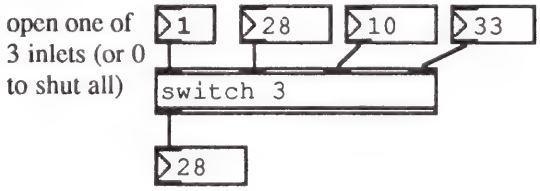


You can see that **gate** can be used to route messages to any of a number of destinations, just by specifying which outlet the messages are to be sent out.

switch

The **switch** object opens one of several inlets to receive messages, and ignores messages received in the other inlets. The leftmost inlet is the control inlet, as with **gate**, and the remaining inlets can receive any message to be sent out the outlet if the inlet is open. The number of inlets—in addition to the control inlet—is specified by the argument. (There can be as many as 9.)

- Send a number to the leftmost inlet of **switch** to open one of the inlets (or 0 to close all inlets). Then send numbers to the other inlets, and you will see that only the one inlet you specified is open.



Using **switch**, you can have messages coming into individual inlets from several different objects, but **switch** "listens to" only one of its inlets.

Any type of message can be passed through **gate** or **switch**.

Tutorial 17

Gates and switches

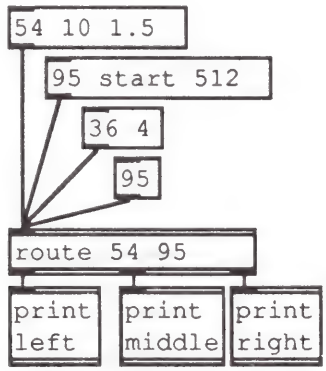
Left Inlet Is Control Inlet

Unlike most objects—which are triggered by a message received in the left inlet—**gate**, **switch**, **Ggate**, and **Gswitch** all use the left inlet as a *control inlet*, for telling which inlet or outlet is to be open. Messages received in the other inlet(s) are then sent out the appropriate outlet.

route

One other valuable traffic controller is **route**, sort of a cross between **sel** and **gate**. When **route** receives a message in its inlet, it compares the first item in the message to each of its arguments. If it finds a match, it sends *the rest* of the message out the corresponding outlet. If there is no match, the entire message is sent out the rightmost outlet.

- Click on the different message boxes and observe what is printed in the Max window.



The number of outlets **route** has depends on how many arguments are typed in. Each argument is an identifier for an outlet, and an additional outlet on the right sends out any messages not matched by **route**.

Like **select**, **route** looks only for certain inputs, and if there is no match it passes the message on unchanged. There are two important differences between **select** and **route**, however.

1. Whereas **select** can have any combination of arguments—int, float, or symbol—the arguments to a **route** object must be either all ints or all symbols.
2) When **select** receives a match for one of its arguments, it sends a bang out the corresponding outlet. When **route** receives a match, it sends out *the rest of the message*. (Unless there is no rest of the message, in which case it sends out bang.)

Summary

The **gate** and **Ggate** objects receive messages in their right inlet, and send the messages out only one of their outlets, depending on which outlet has been specified as *open*. The **switch** and **Gswitch** objects receive messages in only one of their inlets, depending on which inlet has been specified as open.

Tutorial 17

The **route** object tries to match the first item of each incoming message to one of its arguments. If a match is found, **route** sends *the rest* of the message out the appropriate outlet. If there is no match, the entire message is sent out the rightmost outlet.

Routing objects can be used to filter messages coming from many different objects—passing on only those messages which arrive in a particular inlet—or they can be used to send incoming messages to one of many destinations.

See Also

gate	Pass the input out a specific outlet
Ggate	Pass the input out one of two outlets
Gswitch	Receive the input in one of two inlets
route	Selectively pass the input out a specific outlet
switch	Receive the input in a specific inlet

Tutorial 18

Test 3:
Comparisons and decisions

Comparisons and Decisions

You can write a patch that examines the notes being played on your MIDI keyboard, and makes a decision about what to do, based on the note information. Here is an exercise to test your understanding of the use of comparisons to make decisions.

- Make a patch that receives the notes being played on your MIDI keyboard and, if the note is Middle C (MIDI note 60) or higher, plays the notes an octave lower and an octave higher than the played pitch.

Hints

A comparison is needed to find out if the pitch being played *is greater than or equal to* 60.

Based on that comparison, a decision is made to play or not to play the notes an octave lower (12 semitones less) and an octave higher (12 semitones more) than the original pitch.

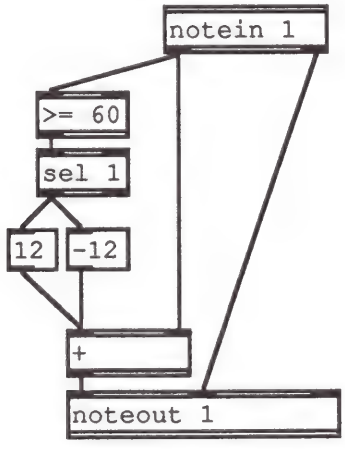
If the original pitch is less than 60, nothing needs to be done, but if the pitch is greater than or equal to 60, two new pitches need to be generated. The original pitch must have 12 subtracted from it to get the lower pitch, and it must have 12 added to it to get the higher pitch.

The new pitches must be combined with a velocity (perhaps the velocity of the original played note) to send note-on and note-off messages to the synth.

- Scroll the Patcher window to the right to see two possible solutions to the problem.

Solution 1

Both solutions use the relational operator **>=** to find out if the played pitch is greater than or equal to 60. Each solution uses the result of that comparison in slightly different ways.



In Solution 1, a successful comparison results in a decision to trigger the numbers -12 and 12. The pitch is stored in the right inlet of a **+** object before the comparison is made. Then, if the pitch is

Tutorial 17

The **route** object tries to match the first item of each incoming message to one of its arguments. If a match is found, **route** sends *the rest* of the message out the appropriate outlet. If there is no match, the entire message is sent out the rightmost outlet.

Routing objects can be used to filter messages coming from many different objects—passing on only those messages which arrive in a particular inlet—or they can be used to send incoming messages to one of many destinations.

See Also

gate	Pass the input out a specific outlet
Ggate	Pass the input out one of two outlets
Gswitch	Receive the input in one of two inlets
route	Selectively pass the input out a specific outlet
switch	Receive the input in a specific inlet

Tutorial 18

Test 3:
Comparisons and decisions

Comparisons and Decisions

You can write a patch that examines the notes being played on your MIDI keyboard, and makes a decision about what to do, based on the note information. Here is an exercise to test your understanding of the use of comparisons to make decisions.

- Make a patch that receives the notes being played on your MIDI keyboard and, if the note is Middle C (MIDI note 60) or higher, plays the notes an octave lower and an octave higher than the played pitch.

Hints

A comparison is needed to find out if the pitch being played *is greater than or equal to* 60.

Based on that comparison, a decision is made to play or not to play the notes an octave lower (12 semitones less) and an octave higher (12 semitones more) than the original pitch.

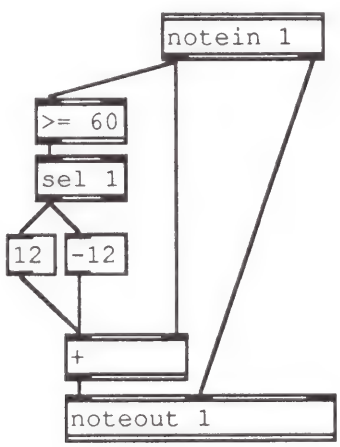
If the original pitch is less than 60, nothing needs to be done, but if the pitch is greater than or equal to 60, two new pitches need to be generated. The original pitch must have 12 subtracted from it to get the lower pitch, and it must have 12 added to it to get the higher pitch.

The new pitches must be combined with a velocity (perhaps the velocity of the original played note) to send note-on and note-off messages to the synth.

- Scroll the Patcher window to the right to see two possible solutions to the problem.

Solution 1

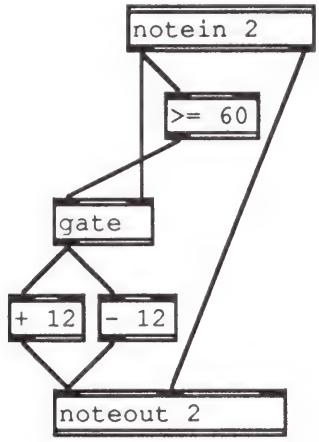
Both solutions use the relational operator **>=** to find out if the played pitch is greater than or equal to 60. Each solution uses the result of that comparison in slightly different ways.



In Solution 1, a successful comparison results in a decision to trigger the numbers -12 and 12. The pitch is stored in the right inlet of a **+** object before the comparison is made. Then, if the pitch is

greater than or equal to 60, the **>=** object sends out 1, causing the **sel** object to trigger the messages -12 and 12. These two numbers are added to the original pitch, and the sums are sent to **noteout**, where they are combined with the velocity of the played note, triggering note messages to the synth. This process occurs both for note-ons and note-offs, so the transposed pitches are successfully played and turned off.

In Solution 2, a successful comparison opens a **gate**, letting the pitch through, so that only pitches greater than or equal to 60 will be passed on.



The placement of the **>=** object is of utmost importance. Because it is to the *right* of the **gate** object, it receives the pitch first, and either opens or shuts the **gate** before the pitch arrives at the **gate**.

When a pitch is let through, it has 12 subtracted from it in one object, and has 12 added to it in another object. The results are sent to **noteout** as pitches, where they are combined with the original played velocity (both for note-ons and note-offs).

Note: The **notein** and **noteout** objects in Solution 2 are set to receive and transmit on MIDI channel 2 only so that a note from your MIDI keyboard won't cause *both* patches to react.

Summary

In this exercise, the comparison **>=** 60 was used to trigger messages in one case, and to open a **gate** in another case. Either method could be incorporated in a solution to the problem.

To perform a test and make a decision, ask yourself these questions:

1. What do I need to know to make the decision? (What will be tested?)
2. What action will be taken if the test is successful (true)?
3. What action will be taken if the test is unsuccessful (false)?

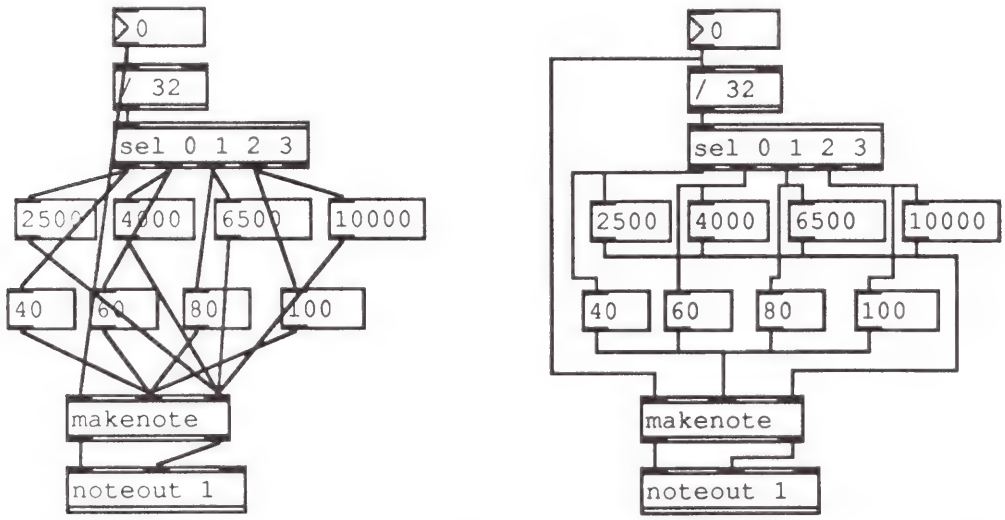
Tutorial 19

Screen aesthetics

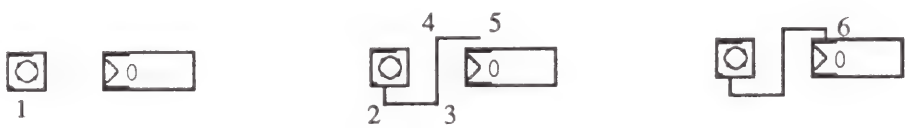
Segmented Patch Cords

So far we have been making patch cord connections with straight patch cords, by dragging from the outlet of one object to the inlet of another object. In complicated patches, though, it would be nice if we could bend patch cords around objects, to keep things from getting too messy.

When **Segmented Patch Cords** is checked in the Options menu, patch cords can be made up of as many as 6 line segments, allowing them to bend around objects. Segmented patch cords function no differently from straight patch cords while making a patch neater and more comprehensible.



The method of connecting objects is a little different when the **Segmented Patch Cords** option is in effect. Instead of dragging from outlet to inlet, the method is to click on the outlet, then click at each of the points where you want the patch cord to bend, then click on the inlet of the receiving object.



The patch in the left part of the Patcher window shows how segmented patch cords can be used to give the patch a neater look, making it easier to understand how the patch functions.

Note that you don't need to turn the Segmented Patch Cords option on in order to make patch cords that turn corners. Shift-clicking on an outlet will allow you to connect a patch cord in "segmented" mode.

Hide On Lock

- Unlock the Patcher window. You will see several objects and patch cords that were not visible before.

Screen aesthetics

Tutorial 19

When editing a patch, you can select objects and/or patch cords and choose **Hide On Lock** from the Max menu. This sets those objects to be invisible when the Patcher window is locked. **Show On Lock** makes objects visible which were previously made invisible with **Hide On Lock**.

Hide On Lock is an invaluable feature for making patches with a good user interface. For example, in the patch in the right part of the Patcher window, there is no reason that the user needs to see the **pgmout** object and the patch cords connecting the various objects. All that the user of your patch needs or wants to see is the **dial**, the label (**comment**), and the number being sent out by the **dial**.

The Hide On Lock command lets you hide unsightly parts of a patch

Note: If you select a region of the Patcher window and hide several objects at once with **Hide On Lock**, the objects will be hidden but the patch cords will still be visible. To hide patch cords, you must select them before choosing **Hide On Lock**. You can select patch cords by holding down the option key while dragging a box around a group of objects. Alternatively, choosing **Select All** from the Edit menu selects all boxes *and* patch cords, or you can just click on patch cords individually, using the shift key to select multiple objects.

Graphical user interface objects (such as **button**, **toggle**, **dial**, etc.) which have been hidden with **Hide On Lock** do not respond to clicking and dragging with the mouse.

Paste Picture

Your completed patch can actually have *any* visual appearance you choose, because you can import pictures from other applications such as painting and drawing programs. Cut or Copy the picture from another application, then in Max, choose **Paste Picture** from the Edit menu to paste the picture into a Patcher window.

For example, the decorative border around the **dial** in this patch is actually just a frame drawn in a painting program. The **dial** and the frame were then placed on top of each other to give the illusion of a different kind of **dial**.

Tutorial 19

Screen aesthetics

Similarly, the two buttons marked OFF and ON are not actual Max objects. They are pictures that were drawn in another program.

Clicking on a Picture

- Lock the Patcher window and click on the picture of a button marked ON. The **metro** object is started, just as if you had clicked on the **toggle**. Click on the picture of an OFF button to stop the **metro**.

We know that the OFF and ON buttons are just pictures, so how do they turn the **metro** off and on? The pictures seem to respond to a mouse click because transparent buttons—**ubutton** objects—have been placed on top of them.

ubutton

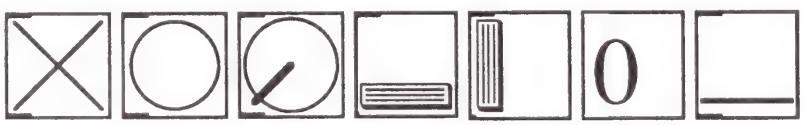
The **ubutton** object is a rectangular button that becomes invisible when the Patcher window is locked. When you click on a **ubutton**, it sends a bang out its right outlet and inverts the part of the screen it covers. When the mouse button is released, **ubutton** sends a bang out its left outlet and becomes transparent again.

A **ubutton** can be placed over a picture or a **comment** (or over nothing, for that matter, if you just want an invisible button) to make that portion of the screen respond to a mouse click. The pixel-inverting feature of **ubutton** can also be used to highlight a spot on the screen. Look under **ubutton** in *Objects* for details.

The connections between the **ubutton** objects and the **toggle** are hidden from sight with the **Hide On Lock** command.

Resizing Objects

The size of objects can be altered by dragging on the *grow bar* in the bottom-right corner of the object. This lets you customize the look of the graphical objects in your Max patches.



Different objects, resized to be the same size and shape

Screen aesthetics

Tutorial 19

Summary

The **Segmented Patch Cords** option lets you create patch cords that bend around objects, making your patches easier to understand. You can also create segmented patch cords without this option turned on by shift-clicking on an outlet.

Objects and patch cords can be hidden from the sight of the user with the **Hide On Lock** command, so that the user sees only those things you want seen. Objects which have been hidden with **Hide On Lock** do not respond to clicking and dragging with the mouse.

Pictures can be imported from graphics applications and placed in a Patcher window with the **Paste Picture** command.

Graphical objects can be resized by dragging on the *grow bar* in the bottom-right corner of the object box. The appearance of a graphical object can also be altered by pasting a picture around it to serve as a frame.

The transparent button object, **ubutton**, lets you make any portion of the screen seem to respond to clicks from the mouse.

The combination of these features lets you design the screen to have almost any imaginable appearance and respond to the mouse in a variety of ways.

See Also

Menus	Explanation of commands
ubutton	Transparent button, sends a bang

Tutorial 20

Using the Macintosh keyboard

ASCII Objects

When you type a key on your Macintosh keyboard, a message is sent to the computer telling it which key you typed. Max has objects for receiving and interpreting this information.

The *American Standard Code for Information Interchange* (ASCII) is the standard system of key numbering. The **key** object receives *key down* messages from the Mac keyboard and sends the ASCII number of the typed key out its left outlet. (Because **key** receives its input directly from the Mac keyboard, it has no inlet.) The ASCII number can then be used in a patch just like any other number.

The **keyup** object is similar to **key**, but it sends out the ACSII number of a key when it is *released* (when a *key up* message is received from the keyboard). The **numkey** object receives ASCII numbers from a **key** or **keyup** object, and deciphers them to determine if a *number* is being typed in on the keyboard. It reports the value of the number the user is entering.

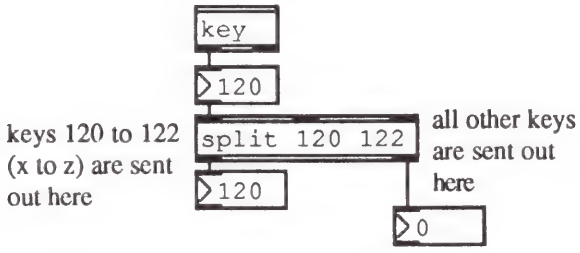
key

This example contains a single, rather complex patch. At the top of Part A is a **key** object. Every time you press a key on the Mac keyboard, **key** sends the ASCII number of that key out its left outlet. The ACSII number is sent to an object called **split**.

split

The **split** object is a combination of relational operator and gate. It looks for a specific range of numbers. If the number received in its inlet is within the specified range, it is sent out the left outlet. Otherwise it is sent out the right outlet.

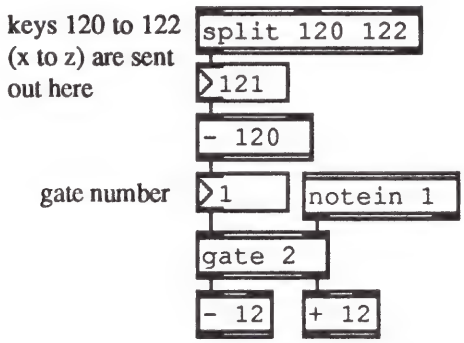
The minimum and maximum values of **split**'s range can be typed in as arguments and/or they can be supplied in the middle and right inlets. In this case, ASCII numbers 120 through 122 (keys *x*, *y*, and *z*) are sent out the left outlet, and all other numbers are sent out the right outlet.



Using Key Commands to Control a gate

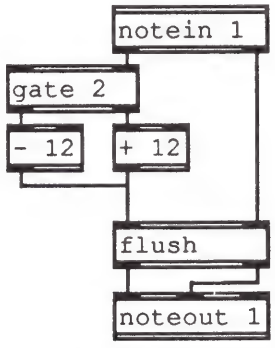
Let's see what happens when the keys *x*, *y*, and *z* (120, 121, and 122) are typed. First of all, 120 is subtracted from the key number, resulting in the numbers 0, 1, and 2. These numbers are used to control a **gate**. The letter *x* closes the **gate**, the letter *y* opens the left outlet, and the letter *z* opens the right outlet.

Tutorial 20



Pitch information from **notein** is passed throught the open outlet of the **gate**. So, if the letter *y* is typed, pitches are passed to the - 12 object and are transposed down an octave. If the letter *z* is typed, the pitches are passed to the + 12 object and are transposed up an octave. If x is typed, the **gate** is closed and no pitches are passed through.

The transposed pitches are combined with velocities in the **flush** object, and are sent to **noteout**.



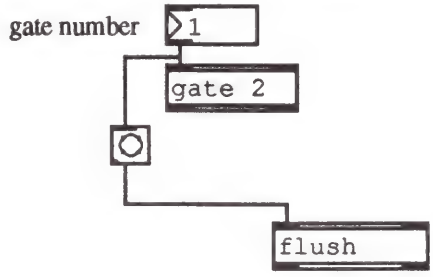
- Type the different key commands *x*, *y*, and *z*, and listen to the change in effect for each command when you play on your MIDI keyboard.

Turning Off Transposed Notes

At first glance, the **flush** object may seem to be unnecessary in this patch. Why can't the pitches be combined with velocities right in **noteout**? They can, but this would leave open the possibility that some note-off messages will not be received by the synth. Consider the following scenario.

Suppose you type the letter *y* to transpose pitches down an octave. Then you play and hold down the note C3 (60) on your MIDI keyboard. This will cause the note 48 to be sent to **noteout**. Before you release the note 60 on your keyboard, you type *z* to transpose pitches up an octave. Now when you release the note 60, a note-off for note 72 will be sent to the synth. The note 48 will not get turned off.

To solve this potential problem, the notes are sent first to a **flush** object. Each time a number is received in the left inlet of **gate**, a bang is also sent to **flush** to turn off any held notes. In this way, note-offs are always provided for any notes that are being held when the status of **gate** is changed.

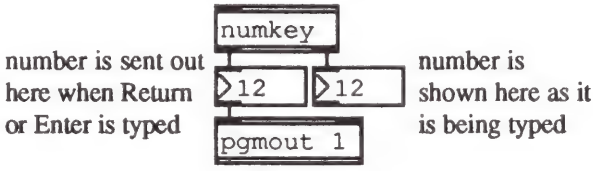


Whenever a **gate** *control number is received, a* bang *is sent to* **flush**

Of course, if you never play notes and give commands at the same time, this precaution is unnecessary. As a general rule, though, whenever you are processing notes (for example, transposing them) it's good to make sure that a note-on message is always followed by a corresponding note-off message. Changing the transposition, closing a gate, etc. while a note is being played can often cause this sort of problem.

numkey

Part B of the patch shows how **numkey** interprets numbers typed on the Mac keyboard. ASCII values from the **key** object are sent to **numkey** (except for the keys *x*, *y*, and *z*, which **numkey** ignores anyway because they are not numerical digit keys).



As digits are typed, **numkey** sends the current number out its right outlet. The Delete (Backspace) key can be typed to erase the last digit of the number. When you have typed in the complete number, you can type the Return key or the Enter key to send the number out the left outlet. In general, the right outlet is used for showing what number is being typed, and the left outlet is used for actually sending it.

- Try changing the sound on your synth by typing in the program number on the Mac keyboard.

Using a combination of **key** and **numkey** to type in numbers is different from typing numbers directly into a **Number box**, because you have to click on the **Number box** before typing into it. The **key** object receives *all* typed keys, so there is no need to select any object before numbers are typed in via **numkey**.

Tutorial 20

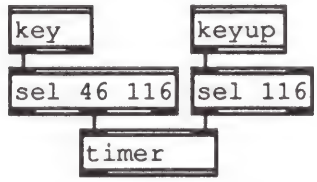
keyup

Typing a key on the Macintosh keyboard, just like playing a note on your MIDI keyboard, sends two messages to the computer—one when the key is pressed down, and another when the key is released. The **keyup** objects sends the ASCII number of any key which is released.

In Part C of the patch, we measure the time that a key is held down by measuring the time between arrival of a number sent by **key** and a number sent by **keyup**.

timer

In Part C, we use **sel** objects to look for specific ASCII numbers. Both **sel** objects look for the number 116, which is the key *t*. (We chose *t* as a mnemonic for *tempo*.) When *t* is pressed down a bang is sent to the left inlet of a **timer** object. When the key *t* is released a bang is sent to the right inlet of the **timer**.



The **timer** object outputs the number of milliseconds between a bang received in its left inlet and a bang received in its right inlet.

Note: **timer** is an exception to the general rule of the left inlet being the one that triggers output. In this case, a bang in the left inlet starts the timing process, but the right inlet is the one that causes the elapsed time to be sent out.

Using Duration to Set Tempo

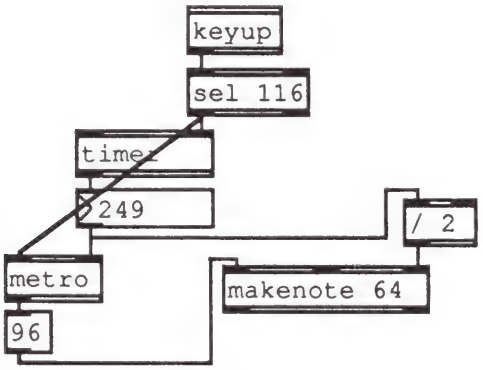
This patch uses the duration that a key is held down to set the speed of a metronome. When the key *t* is released, a bang is sent to the right outlet of **timer**, which reports the time that the key was held down. (We used the message from **key** to start the **timer**.)

The time is sent to the right inlet of **metro**, and is also divided by 2 and sent to the right inlet of **makenote**. In this way, the duration of the notes played will be $1/2$ the amount of time between notes, giving a *staccato* effect.

The release of the *t* key also starts the **metro**. Notice that the **timer** is triggered *before* the **metro**, so that the time values will arrive in **metro** and **makenote** before **metro** is started.

Tutorial 20

Using the Macintosh keyboard



The **metro** sends the pitch 96 to **makenote** until the period key (.) is typed to stop the **metro**.

- Hold the *t* key down for various lengths of time and listen to the change in the tempo of the **metro** (and the duration of the notes). Type the period key (.) to stop the **metro**.

Summary

The **key** object reports the ASCII code of keys typed on the Macintosh keyboard. The **keyup** object reports the ASCII code of keys when they are *released*. The **numkey** object interprets ASCII received from **key** or **keyup**, and reports any numberical values being typed.

ASCII values from **key** or **keyup** can be used to send commands to a patch, opening a **gate** or triggering processes. A relational operator such as **sel** can be used to look for certain keys being typed.

The **split** object looks for numbers within a certain range. If an incoming number is within range, it is sent out the left outlet, otherwise it is sent out the right outlet.

The elapsed time between any two events can be reported with **timer**. The **timer** is started by a bang received in its left inlet, and the elapsed time is sent out when a bang is received in the right inlet.

See Also

key	Report keys typed on the Macintosh keyboard
keyup	Report keys released on the Macintosh keyboard
numkey	Interpret numbers typed on the Macintosh keyboard
split	Look for a range of numbers
timer	Report elapsed time between two events

Storing numbers

Tutorial 21

Variables

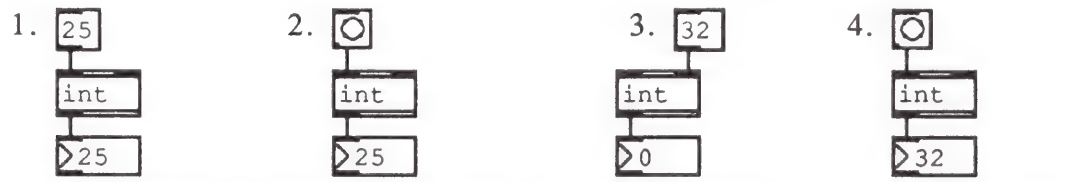
In traditional programming languages, *variables* are places in memory used by a program to store numbers so they can be recalled at a later time.

Many objects in Max are capable of storing a number and recalling it later. For example, the **Number box** will send out the number stored in it when it receives a bang message.

In this tutorial, we'll use objects which do nothing but store a number and send it out when a bang is received in the left inlet. These objects are **int** for storing integer numbers, and **float** for storing decimal numbers.

int and float

When a number is received in the left inlet of an **int** or a **float** object, it is stored and sent out the outlet. Whenever a bang is received in the left inlet, the stored number is sent out again.



the number is stored and is also sent out

bang causes the stored number to be sent out

the number is stored but nothing is sent out

bang causes the stored number to be sent out

When a number is received in the right inlet, it is stored without triggering any output (replacing the previously stored value).

The **int** and **float** objects both function in exactly the same way. The only difference is the type of number they store. When a decimal number (float) is received by an **int** object, the number is converted to an int before being stored, and vice versa.

An initial value to be stored in **int** or **float** can be typed in as an argument. If there is no argument, the objects intially store the number 0.

Using float

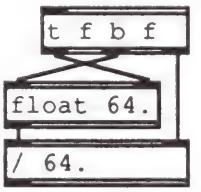
The patch in the left part of the Patcher window uses both **int** and **float**. Once we understand what the patch does, the need for these objects should be clear.

Tutorial 21

Storing numbers

The combination of **notein** and **stripnote** should be familiar to you. It's the easiest method of getting note-on data from a MIDI keyboard (without getting note-off data).

The velocity is converted to three separate messages by the **t** (**trigger**) object: a float, a bang, and another float. The first float (from the right outlet of **t**) is sent to the right inlet of the * object, where it is stored. The bang then causes the **float** object to send out its stored number.

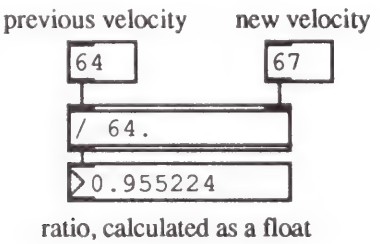


This triggers a series of calculations, finally resulting in a number being sent to the right inlets of **makenote** and **metro**. Then the last float (from the left outlet of **t**) is stored in the **float** object.

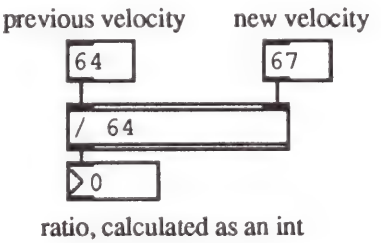
What this means is that each time a new velocity is received, the *previous* velocity is divided by the *new* velocity, then the new velocity is stored as the "previous" one. The result is the ratio of the old velocity to the new velocity.

Note: The velocity could just as well be stored in an **int** object, but it is converted to float by the / object in any case. Since the conversion from int to float has to occur somewhere, we made the conversion with the **t** object.

Why do we need to divide the two velocities as floats? Consider the possible cases. The range of possible ratios between two note-on velocities is from $^{1}/_{127}$ to $^{127}/_{1}$, (i.e., from 0.007874 to 127.0). Whenever the previous velocity is less than the new one, the ratio will be some fraction between 0 and 1. But if we performed an integer division, the result would always be 0 when the velocity is increasing.



ratio, calculated as a float



ratio, calculated as an int

Float division is needed to get the precise ratio between two velocities

Tutorial 21

The ratio is then multiplied by 250 (again the numbers are calcualted as floats, so that ratios less than 1 are not converted to 0 before the multiplication), and the result is used as the new note duration for **makenote** and the new tempo for **metro**.

However, not all the numbers we get this way are really suitable as musical values. Extreme changes in velocity result in very small or very large numbers. The range of possible values in this calculation can be as small as 1.9685 (which will be truncated to 1 when it is converted to int) and as great as 31750. So, we use **split** to limit the values between 40 milliseconds (25 notes per second) and 2000 milliseconds (one note every 2 seconds). Numbers which exceed these limits will be ignored. The **split** object automatically converts the floats back to ints.

- Play a few notes on your MIDI keyboard, and observe how changes in velocity affect the numbers being sent out of **split**. When the velocity is increasing, the numbers are less than 250. When the velocity is decreasing the numbers are greater than 250. Extreme changes in velocity result in extremely large or small numbers, which are ignored by **split**.

Using int

What happens next in our patch? The velocity is sent to the middle inlet of **makenote**, where it is stored. Then the pitch value is stored in the **int** object.

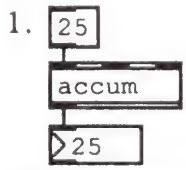
Well, so far we've seen that a lot of numbers get stored, calculated, and changed by playing on the synth, but nothing else happens...until we turn on the **metro**. The bangs from the **metro** trigger the **int** object, which sends out its number—whatever pitch was most recently played—to **makenote**. The speed of the **metro** is dependent on the change in velocity between successive notes played on the synth.

- Turn on the **metro** and play on your MIDI keyboard. Notice how you can affect the speed, velocity, and duration of the repeated notes by changing the velocity with which you play.

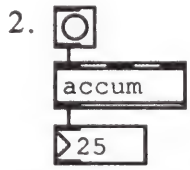
accum

Another storage object, **accum**, performs internal additions and multiplications to change its stored value.

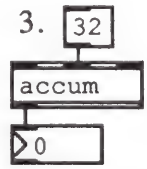
The left inlet of **accum** functions just like that of the **int** and **float** objects: a number received in the left inlet is stored and sent out the outlet, and a bang sends the stored number out again. However, the middle and right inlets of **accum** are used to *add to* the stored number or *multiply* the stored number, respectively. The number is changed without anything being sent out the outlet.



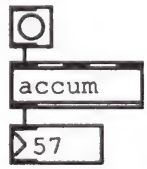
the number is stored and is also sent out



bang causes the stored number to be sent out



the number is added to but nothing is sent out



bang causes the stored number to be sent out

Tutorial 21

Storing numbers

An initial value to be stored in **accum** can be typed in as an argument. If there is no argument, the object intially stores the number 0. The value stored in **accum** is normally an int, but if the typed-in argument contains a decimal point **accum** stores a float. Multiplication in **accum** is always done with floats, even if the stored number is an int.

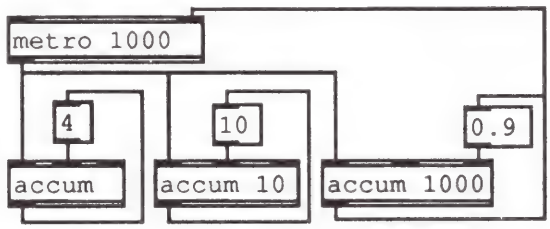
Using accum

The **accum** object is most useful for storing a value that you wish to change often by adding to it or multiplying it. For example, you may want to continually *increment* a number by adding some amount to it over and over. The second patch in the Patcher window shows an example of incrementing.

- Click on the **toggle** to start the **metro**. Notice how the pitch, velocity, and duration values sent to **makenote** change continually. The amount of change is directly related to the numbers being added or multiplied in the **accum** objects.

Each time the **accum** objects receive a bang from the **metro**, they send their stored values to **makenote**, and they also trigger message boxes which add some number back into the stored value or multiply the stored value by some amount. (Note: Until now we've usually triggered the message box with a bang, but it can also be triggered with a number.)

The result is that the **accum** objects increase in value each time they receive a bang.



These values would soon exceed reasonable ranges unless we place some kind of restriction on them. In this patch, numbers sent to **makenote** loop repeatedly through cycles of different lengths. Two methods of looping are shown.

The **accum** that sends durations to **makenote** (and tempos to **metro**) starts at 1000, and multiplies itself by 0.9 every time it sends out a number. Eventually the number is reduced to be less than 40. When this happens, the message set 1000 is sent to the left inlet of **accum**.

We have already seen a set message used to set the value of a **slider** without triggering output. It has the same effect when sent to the left inlet of **accum**. Every time the value of the **accum** goes below 40, it is reset to 1000 and the cycle begins again.

The tempo of the **metro** is also set back to 1000 at the same time, by sending a bang to the **1000** object. An object box that contains only a number is actually an **int** object (or a **float** object if the number contains a decimal point) with an initial value of 1000.

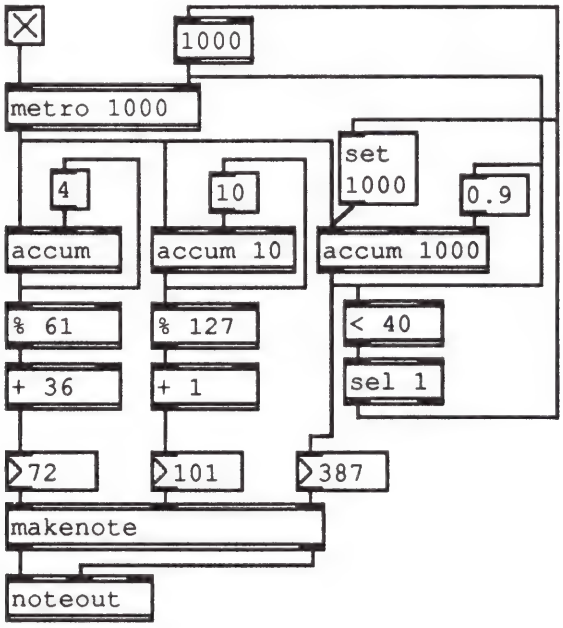
Tutorial 21

This shows the basic method of looping: change a value continually until some condition is met (for example, until it exceeds some limit), then reset the value and begin again.

The values stored in the other two **accum** objects continue to increase without being reset, but the modulo operator **%** limits the numbers so that they always cycle within a limited range. Using the **%** object is another good way of looping.

Loops Can Create Cyclical Patterns

Using loops in a program is a way of creating *periodicities*. In this patch, the parameters of duration, velocity, and pitch all have a different periodicity of recurrence, which makes the music seem to repeat itself while always changing slightly.



Actually, the duration changes according to a 32-note cycle, the velocity changes according to a 118-note cycle, and the pitch changes according to a 37-note cycle. Thus, the entire pattern is repeated every 69,856 notes—about every 6 hours. Not a melody likely to get stuck in your head.

Overdrive

Max can get rather busy playing music, running metronomes, doing mathematical calculations and printing numbers on the screen. Checking **Overdrive** in the Options menu causes Max to give priority to music-making tasks, and results in more accurate musical timing. If you hear improper delays on notes, or erratic performance, try enabling **Overdrive**.

Tutorial 21

Storing numbers

Summary

Integer numbers can be stored in an **int** object, and sent out later by triggering **int** with a bang in its left inlet. Decimal numbers can be similarly stored and recalled with a **float** object. The **accum** object can store and recall either an int or a float value, and can add to or multiply the stored value without sending it out.

Floats are useful for calculations which involve numbers between 0 and 1, or for any calculation which requires additional precision.

The **split** object is useful for limiting numbers to a specific range. All numbers it receives which are within its specified range are sent out its left outlet. Otherwise, they are sent out its right outlet.

Cycles of numbers can be produced by *looping*. A loop is created by continually changing a number, then resetting it when the number meets a certain condition. The **accum** object is well suited for such looping schemes.

See Also

accum	Store, add to, and multiply a number
float	Store a decimal number
int	Store an integer number
Loops	Using loops to perform repeated operations

Delay lines

Tutorial 22

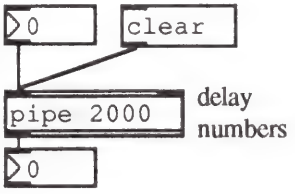
Delaying Numbers and bangs

Messages normally pass through patch cords as fast as the computer can send them. However, you can also delay numbers, lists, or bangs for a certain amount of time before sending them on their way. This is useful for creating a specific time lag between messages, or delaying notes to create echo effects.

The **pipe** object delays the numbers it receives for a certain amount of time before sending them on. The **delay** object (also called **del**) delays a bang for a certain amount of time before sending it on.

pipe

- Drag on the **Number box** at the top of Patch 1. The numbers are all delayed for 2000 milliseconds by the **pipe** object.

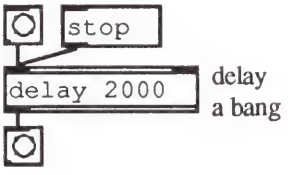


Each number is sent out the outlet a certain amount of time after it is received, so **pipe** can delay many numbers and send them out later in the same rhythm in which they were received. A number received in the right inlet sets the delay time, in milliseconds, to be applied to all numbers subsequently received in the left inlet. A clear message received in the left inlet erases any numbers currently being delayed in the **pipe**.

- Send some more numbers to the left inlet of the **pipe** in Patch 1, then quickly click on the word clear. Any numbers not yet passed through the **pipe** are forgotten.

delay

- Click on the **button** at the top of Patch 2. The bang is delayed for 2000 milliseconds before being sent out the outlet of the **delay** object.



Unlike **pipe**, which can keep track of many numbers at a time, **delay** can keep track of only *one* bang at a time. If **delay** receives a new band while it's already delaying a bang, the old bang is lost and the new bang is delayed instead.

Tutorial 22

Delay lines

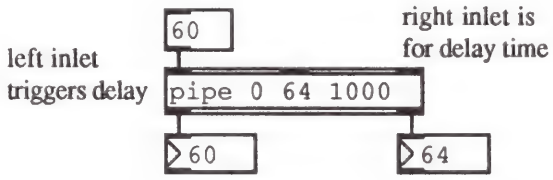
- Send many bangs to **delay** in quick succession. Each bang received within 2 seconds of the previous one *erases* the previous one, so only the last bang gets sent out the output.

A number received in the right inlet of **delay** sets the delay time to be applied to any bang subsequently received in the left inlet. A stop meassage received in the left inlet stops any bang currently being delayed.

- Send another bang to **delay**, then quickly click on the word stop. The bang is not sent out.

Delaying Groups of Numbers

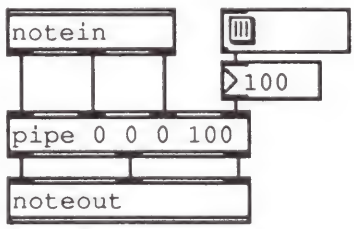
A single **pipe** object can actually delay several parallel streams of numbers—such as pitch, velocity, and channel information from **notein**—with a separate pair inlets and outlets for each stream. To make a **pipe** with more than one outlet, type in one number argument for each stream of numbers you want to delay, plus an argument for the delay time. The last argument is always the delay time.



Arguments set intial value for each delay line.
Last argument is the delay time.

As with most objects, it's the left inlet that triggers the **pipe**. When a number is received in the left inlet, it is delayed along with whatever number was most recently received in the other delay line inlets. If no number has been received yet in the other inlets, the initial value that was named in the argument is used, as in the example above. The numbers can also be received together as a list in the left inlet, with an additional number included at the end of the list to specify the delay time.

Patch 3 shows a **pipe** that delays three streams of numbers. The channel and velocity values from **notein** are received, and then the pitch value triggers the delay of all three numbers. The delay time can be changed by sending a new number to the right inlet of **pipe**.



- Try playing on your MIDI keyboard using different delay times.

Tutorial 22

random

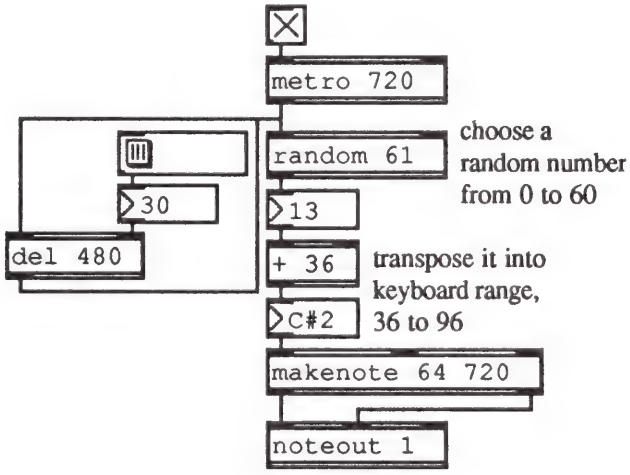
Each time the **random** object receives a bang in its left inlet, it chooses a number at random and sends the number out the outlet. The range of numbers from which **random** chooses is determined by typing in an argument or by sending a number in the right inlet. The random values will always be between 0 and one less than the argument.

Different uses of random numbers will be seen in the course of the Tutorial.

Using Delayed Triggers

In Patch 4 the **metro** triggers **random** to send out a random number between 0 and 60 once every 720 milliseconds. 36 is added to the number to bring it up into the standard MIDI keyboard range, and it is then transmitted as a pitch to be played on the synth.

The bang from **metro** is also sent to the **del** object, where it is delayed a certain time before being sent to **random**. A new randomly chosen number is then sent out, so actually two notes are played every 720ms. The rhythm between the two notes depends on the delay time sent to the right inlet of **del** from the **hslider**.



- Turn on the **metro** and experiment with creating different rhythms by changing the delay time of the **del** object.

Tutorial 22

Delay lines

Summary

A single bang can be delayed for a specific amount of time by the **delay** object, also called **del**. If a second bang is received while the first bang is being delayed, the first bang is forgotten and the second bang is delayed.

Numbers or lists can be delayed by the **pipe** object. A **pipe** can delay a series of numbers, and output them later in the same rhythm in which they were received.

A **pipe** can also delay a list of numbers (or numbers received together, such as pitch-velocity pairs) when arguments are typed in to indicate how many numbers are to be delayed.

The delay time is specified in milliseconds, by a number received in the right inlet (or typed in as an argument).

Delays can be used to create echo effects or rhythms.

Each time the **random** object receives a bang in its left inlet, it generates a random number between 0 and one less than its argument and sends the number out its outlet.

See Also

delay	Delay a bang before passing it on
pipe	Delay numbers or lists
random	Output a random number

Test 4:
Imitating a performance

Tutorial 23

Creating Imitation

In previous chapters you have seen how to transpose notes played on your MIDI keyboard, and how to delay notes. Try making a patch of your own that imitates what you play, starting on a different note.

1. Make a patch that imitates whatever you play, 3 seconds after you play it, transposed up a perfect fifth, *and* also imitates whatever you play 6 seconds later, transposed up an octave.

Hints

The notes you play on your keyboard will have to be sent to two different places. In one, the pitch will be transposed up by 7 semitones and all the note data will be delayed by 3000 milliseconds. In the other, the pitch will be transposed up by 12 semitones and the note data will all be delayed by 6000 milliseconds.

Each imitation should use one **pipe** object to delay velocity and pitch data together.

- Page once to the right in the Patcher window to see the solution to the problem, labeled Patch 1. Note that the order of the + object and the **pipe** object could be reversed, i.e., the transposition could take place *after* the delay.

Let the User Type In a New Delay Time

If that exercise was too easy for you, try this more difficult one.

2. In Patch 1 each imitation of the melody comes 3 seconds later than the previous one. Make a version of Patch 1 which lets the user type in a new delay time between imitations.

This presents two potential problems:

- What happens if the user types in a ridiculous delay time such as -1000 or 3600000?
- What happens if the user types in a much shorter delay time while holding down a note, and the note-off gets delayed less than the note-on and is played *before* the note-on?

These problems represent the sort of extreme or unlikely cases you must take into consideration to protect against your program malfunctioning or producing unwanted results.

Dealing With Potential Problems

The problem of negative delay times is not serious because the **pipe** object will set any negative delay time it receives to 0. The problem of extremely large delay times, on the other hand, can be more serious.

If the delay time is very long and you are playing a lot of notes, the number of notes being stored could cause **pipe** to run out of memory. This would cause some notes to be lost, and could conceivably even cause the computer to crash.

The way to deal with this problem is to limit the numbers the user types in as a delay time, and only send them to **pipe** if they are reasonable. Try using a **split** object or **hslider** to limit the numbers between 0 and 15000.

The problem of note-offs being played before note-ons could occur if the user types in a much smaller delay time while holding down a note on the synth. It would result in stuck notes on the synth.

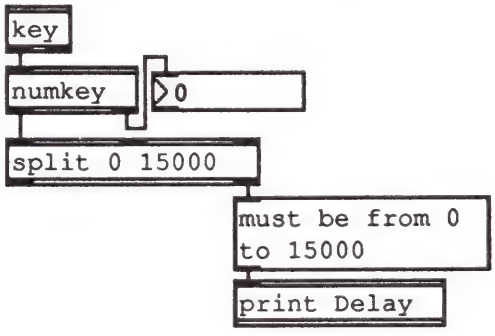
One solution is to compare each new delay time to the previous one. If the new one is smaller, send a note-off to **pipe** for any pitch being held down on the synth. This requires running the notes through a **flush** object before sending them to the **pipes**, and also requires comparing the delay times and sending a bang to the **flush** objects if a smaller delay time is typed in.

Solution to Exercise 2

- Scroll the Patcher window all the way to the right to see Patch No. 2, a possible solution to the exercise.

Note: We have set Patch 2 to receive notes only on MIDI channel 2 so that it will not play while you are trying out Patch 1. To hear Patch 2, set your keyboard to transmit on MIDI channel 2.

We have used a combination of **key** and **numkey** to get the numbers typed on the Macintosh keyboard. The typed numbers are sent to **split**, and any numbers less than 0 or greater than 15000 will cause an error message to be printed in the Max window.



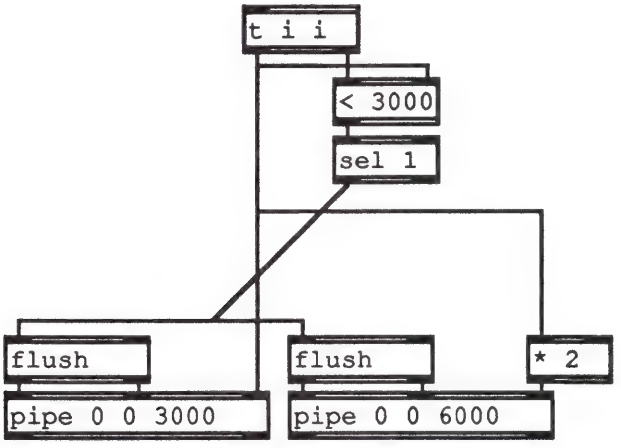
Invalid numbers cause an error message to be printed.

The typed delay time is first sent to the relational operator **<** to compare it with the current delay. If it is less than the current delay, a bang is sent to the **flush** objects, causing them to send out note-offs for any notes that may be held down on your MIDI keyboard.

The new delay time is then sent to the right inlet of **<**, to be stored as the current delay time. It is also sent to the right inlet of the two **pipe** objects. Note that it is doubled before being sent to the **pipe** on the right, so that the right pipe will delay twice as long as the left **pipe**.

Test 4:
Imitating a performance

Tutorial 23



If new delay time is less than the current one
flush any held notes before changing the delay time.

Summary

Delay and transposition can be combined to create imitation.

Always consider unlikely possibilities. For example, whenever you ask the user to supply a value, check to make sure it is a valid value before using it. (You can print an error message when an invalid number is received, or you can just change it to some valid value.) Whenever you are processing note data, make sure that note-ons are always followed by note-offs.

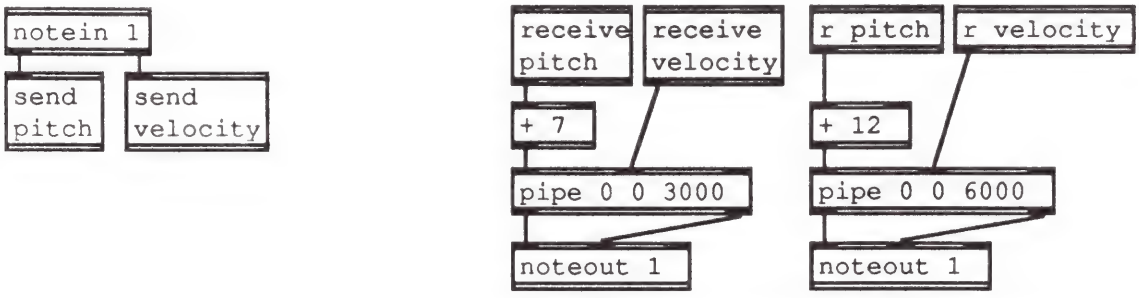
Tutorial 24

send *and* **receive**

Sending Messages Without Patch Cords

It's possible to send any type of message without using a patch cord, with the **send** and **receive** objects. A message in the inlet of a **send** object comes out the outlet of any **receive** object that has the same argument.

In this patch we have redone the imitating patch from the previous chapter using **send** and **receive** objects (also called **s** and **r**).



- Play on your MIDI keyboard. The note data is sent to the **receive** obejcts (and **r** objects) which have the same name (argument) as the **send** object.

The name argument of a **send** object is like a unique radio frequency, and any **receive** object with the same name is "tuned in" to that frequency. Any type of message can be sent with **s** and **r**: ints, floats, lists, symbols, etc.

Communication Between Patcher Windows

The **s** and **r** objects have one particular advantage over patch cords, in that they can communicate *even if the objects are not in the same Patcher window*. This is a very valuable feature, enabling different patches to communicate with each other. Name your **send** and **receive** objects so you won't send a message to another Patcher window unintentionally.

value

The **value** object (also called **v**) stores any int or a float value recieved in its inlet. The number is sent out when a bang is received. All **value** objects with the same argument share the same storage location in the computer's memory, so the number can be stored and recalled by any one of the objects. When a new number is stored in one **value** object, all others which share the same name will also contain the new number.

send *and* **receive** Tutorial 24

A number stored in one location can be recalled in another location.

- Use one of the **Number boxes** to store a number in the **value** object named share. The number can be recalled from any of the **value** share objects by sending a bang to its inlet.

All **value** objects with the same name share the same value, even if they are located in different Patcher windows.

Summary

Any message received in the inlet of a **send** object comes out the outlet of all **receive** objects with the same name (argument), even if they are in different Patcher windows. This is valuable for communicating between Patchers.

A number stored in a **value** object is shared by all value objects with the same name, even if they are in different Patcher windows. When a **value** object receives a bang in its inlet, it sends the number out the outlet (even if the number was received in another **value** object with the same name).

See Also

receive	Receive messages, without patch cords
send	Send messages, without patch cords
value	Share a stored number with other objects

Tutorial 25

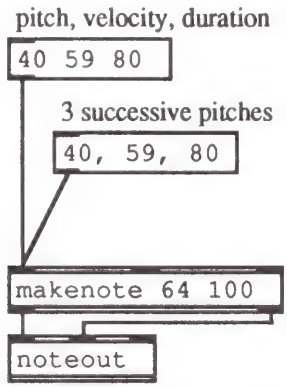
Managing messages

Using the message box

So far we have used the **message box** to send a single message, triggering it either with a mouse click or with a bang, a number, or a list in its inlet. The message box has many additional features for constructing and changing messages, some of which are displayed in this patch.

comma

If a message box contains a comma, messages are sent out one after another. In this way, messages can be sent in rapid succession in response to a single trigger.



- Click on the two message boxes (marked A) in the bottom-left portion of the Patcher window. One message box contains a list of three numbers, 40 59 80. When **makenote** receives the list, it interprets the third number as a duration, the second number as a note-on velocity, and the first number as a pitch. The other message box contains three separate messages. It sends 40, then 59, then 80, and each number is interpreted as a pitch by **makenote**. You can see the messages printed in the Max window, and you can hear the difference in result.

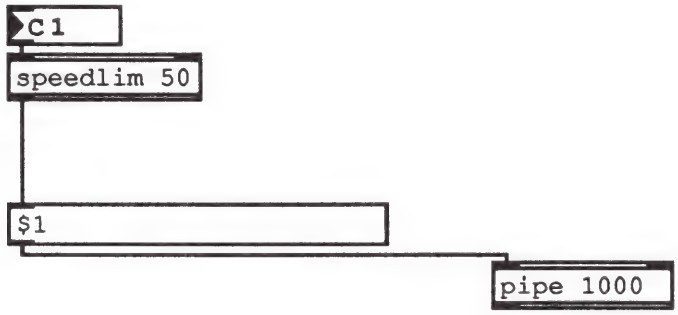
This is one way to play a chord.

The Changeable Argument: $

The dollar sign ($) is a special character in a message box. It is a *changeable argument*, an argument that is replaced by an item from the incoming message. For example, if a message box contains The pitch is $1 and the velocity is $2, and receives the message 60 64 in its inlet, it will send out The pitch is 60 and the velocity is 64. The numbers 60 and 64 are stored in place of $1 and $2 until they are replaced by other values received in the inlet.

- Drag on the **Number box** (marked B) in the top-left corner of the Patcher window. After being limited by **speedlim**, each of the numbers triggers the message box. Because the message box contains the changeable argument $1, the $1 is replaced by the incoming number before the message is sent out.

Tutorial 25



The incoming number is stored in the changeable argument $1 *before the message is sent out.*

The number is then sent to **pipe**, and 1000ms later it is sent to **makenote** and on to the synth.

- The last number to trigger the message box is still stored in place of the $1 argument. Now if you trigger the message box with a bang, the stored number will be sent out again.

The set Message

We have already seen that the message set, followed by a number, can what is stored in many objects without triggering any output. The word set, followed by *any message* can replace the contents of a message box without triggering output. The word set by itself clears the message. (When an empty message is triggered, nothing is sent out.)

- Click on the different set messages in the portion of the patch marked C.

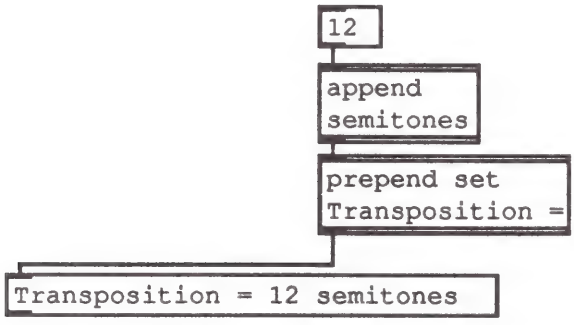
- Although the text in the message box changes, nothing is sent out until it is triggered with the bang message.

append and prepend

The **append** and **prepend** objects are for constructing complex messages. The **append** object *appends* its arguments (preceded by a space) at the end of whatever message it receives, and sends the combined message out its outlet. The **prepend** object places its arguments (followed by a space) before the message it receives, and sends the combined message out its outlet. An example of these objects is in the bottom-right part of the Patcher window.

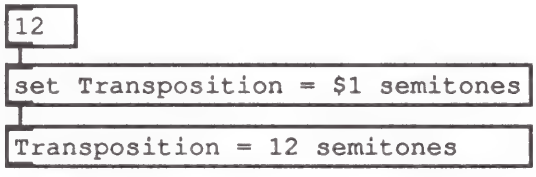
Tutorial 25

Managing messages



When the **append** object receives a message—for example, the number 12— it places the word semitones after it and sends out 12 semitones. The **prepend** object then puts set Transposition = before it and sends out set Transposition = 12 semitones, which changes the message box to Transposition = 12 semitones.

The same result could be obtained using only message boxes, in the following manner:



backslash

The backslash (\) is a special character for negating *other* special characters. A special character which is preceded by a backslash loses its special characteristics and is treated like any other character. This is necessary if you want to include a character such as a comma or a dollar sign in a message without its being interpreted to have a special meaning.

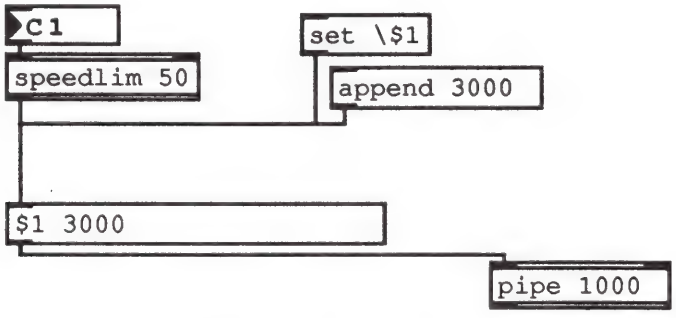
The append Message

When append, followed by any message, is sent to a message box, the message following append will be added to the contents of the message box.

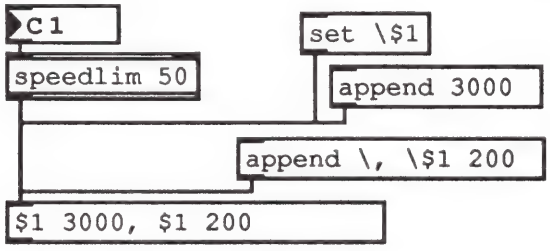
- In the part of the patch marked D, click on the messages set \$1 and append 3000 to construct the message $1 3000. (Notice that we had to precede the dollar sign with a backslash. Otherwise, the message box would have tried to interpret $1 as a changeable argument, and the message would have been set 0). Then drag on the **Number box** marked B.

  The $1 argument is replaced by the incoming number and is sent out as a list with the number 3000. The list is received by **pipe**, 3000 is stored as the new delay time, and the numbers are delayed for 3 seconds before being sent on.

Tutorial 25



- Next, click on all three message boxes in part D, to construct the message **$1 3000, $1 200.**

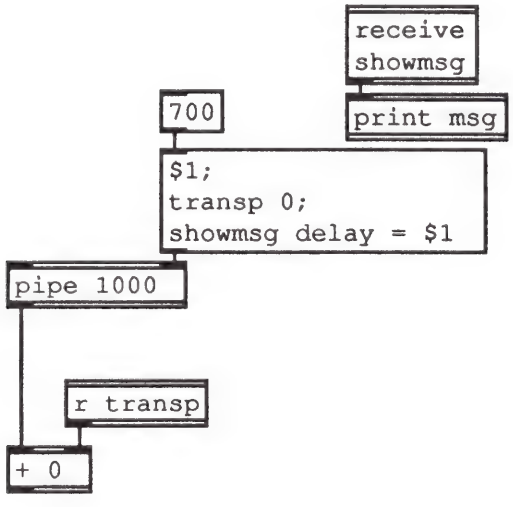


- Now when you send numbers to the message box, it sends out two lists, resulting in each number being delayed 3000ms *and* 200ms.

semicolon

When a semicolon (;) appears in a message box, the first word after the semicolon is interpreted as the name of a **receive** object. The rest of the message (or up to the next semicolon) is sent to all **receive** objects with that name, instead of out the message box's own outlet.

- Click on the message box marked E, containing the number 700.



The number 700 is sent out the outlet to the right inlet of **pipe**, the number 0 is sent out the outlet of the **r** transp object, and the message delay = 700 is printed in the Max window. This is a way of sending many different messages to different places with a single trigger.

Summary

In addition to simply being able to send any message out its outlet, the **message box** can be used to construct messages, and to send them to different places.

The *comma* is used to separate different messages within a message box, and send them out one after the other. When a message is preceded by a *semicolon* in a message box, the first word after the semicolon is the name of a **receive** object, and the rest of the message (or up to the next semicolon) is sent to all **receive** obejcts with that name, instead of out the message box's outlet . The comma and the semicolon enable a message box to send many different messages with a single trigger.

The *dollar sign*, followed immediately by a number (such as $1) is a *changeable argument.* When the message box receives a triggering message in its inlet, each changeable argument is replaced by the corresponding item from the triggering message. ($1 is replaced by the first item, $2 is replaced by the second item, etc.) If no item is present in the incoming message to replace the value of a changeable argument, the previously stored value is used. If no value has yet been stored in a changeable argument, its value is 0 by default.

A *backslash*, used before a special character such as a comma, a semicolon, or a dollar sign, negates the special characteristics of that character.

A set message can be used to change the contents of a message box without triggering any output. An append message can be used to add things to the end of a message box.

The **prepend** and **append** objects attach their typed-in arguments to the beginning or end of incoming messages, then send out the combined message.

See Also

append	Append arguments at the end of a message
Arguments	$ and #, changeable arguments to objects
message	Send any message
prepend	Put one message at the beginning of another
Punctuation	Special characters in objects and messages

The **patcher** *object*

Tutorial 26

Subpatches

A Patcher program can contain other Patcher programs as *subpatches*. The **patcher** object lets you create a patch within a patch.

A new Patcher window opens when you create a new **patcher** object. You can edit a patch in the newly opened *subpatch* window, then when you save your *main* patch, the subpatch is saved as part of the same document. If the subpatch window is open when the document is saved, it will be automatically reopened the next time you open the document. The subpatch window can be brought to the foreground at any time by double-clicking on the **patcher** object. You can even *nest* **patcher** objects; that is, put **patchers** within **patchers**, within **patchers**, etc.

A **patcher** object can be given an argument specifying the name to be shown at the top of the subpatch window. If there is no argument, the window is named *sub patch*. The name is enclosed in brackets to show that it's part of another patch.

This patch contains two **patcher** objects, named *modwheel* and *keyboard*, and their contents are shown in the subpatch windows. For aesthetic reasons we have hidden most of the objects in the subpatches with **Hide On Lock**, but we will examine them shortly.

- Play a few notes on your MIDI keyboard and move the modulation wheel. You will see the **dial** and **kslider** display your actions in the two subpatch windows.

All Windows Active

In Macintosh applications, the front window is the *active window*, where you apply menu commands such as Save and Close, and click and drag on objects. To make a background window active you have to first click on it to bring it to the foreground.

The **All Windows Active** option lets you use background windows without bringing them to the front. To bring any window to the front, click on its title bar or choose its name from the Windows menu, or command-click on any visible part of the window.

- Check **All Windows Active** in the Options menu. This will let you click and drag on the **dial** and **kslider** in the background windows without bringing the windows to the foreground.

- Drag on the **dial** and the **kslider**. They can send data *to* the synth as well as display data received *from* your keyboard.

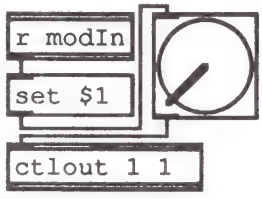
You can close a subpatch window by clicking in its close box, and you can reopen it by double-clicking on the (locked) **patcher** object. Now let's examine the contents of the **patcher** objects.

Tutorial 26

The **patcher** *object*

The modwheel Subpatch

- Bring the *modwheel* window to the foreground and unlock it.

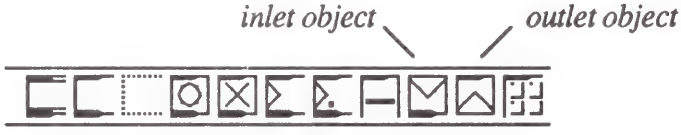


Now you can see the hidden objects. Modulation wheel data received from your MIDI keyboard with **ctlin** is sent to the **r** modIn object in the subpatch. The control data replaces the **$1** argument and sets the **dial** without triggering any output (so the data won't be echoed back to the synth). When you change the **dial** , the data is sent to the synth with **ctlout**.

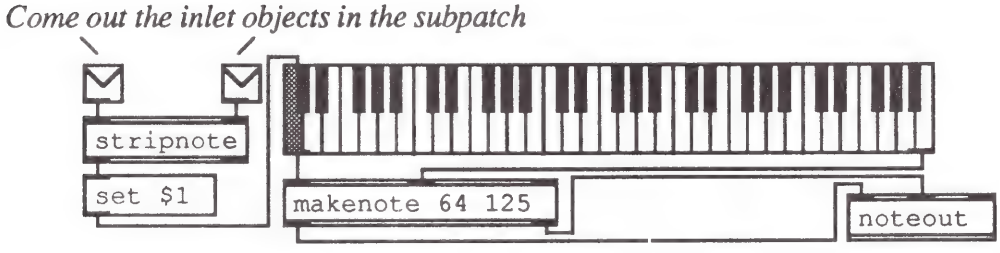
Because they can communicate from one Patcher window to another, **send** and **receive** objects allow you to send messages back and forth between a patch and an embedded subpatch.

inlet and outlet Objects

- Now bring the *keyboard* window to the foreground and unlock it to see the hidden objects. At the top of the subpatch you see two **inlet** objects.



When you include an **inlet** or **outlet** object in a subpatch, a corresponding inlet or outlet is created in the **patcher** object in the main Patcher window. This is usually the most efficient way to send messages to and from a subpatch.



The **patcher** *object*

Tutorial 26

Assistance

When **Assistance** is checked in the Options menu, Max gives you information about the inlets and outlets of objects while you are editing a patch. Every time you place the mouse on an inlet or an outlet, a brief description of that inlet or outlet is printed in the bottom bar of the Patcher window.

You can give Assistance descriptions to the inlets and outlets of your **patcher** object. To do so, select the **inlet** or **outlet** object in your subpatch and choose **Get Info...** from the Max menu. You can type in a description which will show up as an Assistance message when you are working in the main Patcher window.

- Unlock the main Patcher window and pass the mouse over the inlets of the **patcher** keyboard object to see the Assistance messages.

Although writing Assistance messages to yourself may seem like a waste of time, it can be very helpful in reminding you later what type of message a subpatch object expects to receive in its inlet and what type of message will come out of its outlet.

Summary

The **patcher** object creates a *subpatch* within a patch. The subpatch is saved as part of the document that contains the **patcher** object. If the subpatch window is open when the patch is saved, it will be opened automatically when the document is reopened. You can even *nest* a **patcher** object within another **patcher** object.

Messages can be sent between the main patch and the subpatch with **send** and **receive** objects, or with **inlet** and **outlet** objects. When **inlet** or **outlet** objects are placed in a subpatch, corresponding inlets or outlets are automatically created in the **patcher** object.

When **Assistance** is checked in the Options menu, Max prints a description of inlets and outlets in the bottom bar of the Patcher window while you are editing a patch. You can assign Assistance messages to the inlets and outlets of a **patcher** object by selecting the **inlet** or **outlet** object in the subpatch and choosing **Get Info...** from the Max menu.

When **All Windows Active** is checked in the Options menu, you can click and drag on objects in a background window without first bringing the window to the front.

See Also

inlet	Receive messages from outside a Patcher
Menus	Explanation of commands
outlet	Send messages out of a Patcher
patcher	Create a subpatch within a patch

Tutorial 27

Your object

Use Your Patch as an Object

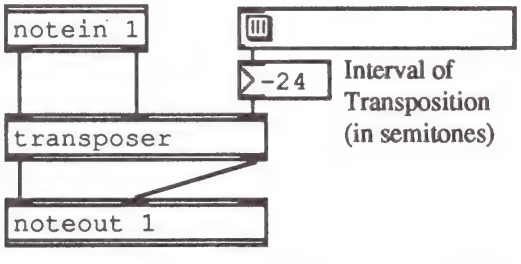
As you get involved with writing your own patches, you will probably find that you are using certain configuations of objects very frequently, or that there are certain computational tasks that you need to do very often. It would be nice if you could just make an object to do that task, then plug in the object wherever necessary.

Actually, any patch you have created and saved can be used as an object in another patch, just by typing the filename of your patch into an object box as if it were an object name.

As we saw with the **patcher** object, when you use a patch within a patch you usually want to be able to communicate with the subpatch. Therefore, when you are making a patch that you plan to use as an object inside another patch, you will usually want to include **inlets** and **outlets** (or **sends** and **receives**) so that you can send messages to your object and it can send messages out.

The transposer Object

In this patch you see a **transposer** object, for transposing incoming pitches and sending out the transposed pitch. The interval of transposition (the number of semitones up or down) is supplied in the right inlet.



The **transposer** is not a built-in Max object. It's a patch that we created and stored in a file named *transposer*.

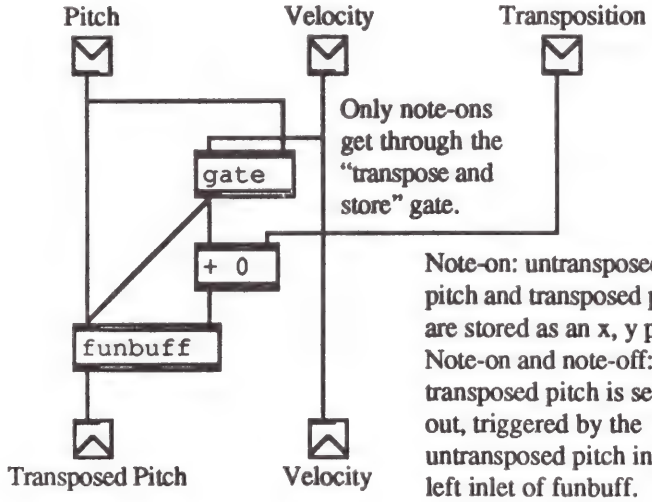
- Double-click on the **transposer** object to see its contents.

In previous patches we have simply sent the pitch to a **+** or **-** object to transpose a note. Why do we need a subpatch like this just to transpose notes? The advantage of the **transposer** over a simple + operator is that the **transposer** ensures that note-offs are transposed by the same interval as their corresponding note-ons, even if the interval of transposition changes while the note is being held down.

If a note-off message is transposed by a different interval than its note-on was transposed, the note-on will never get turned off and the note will be stuck on the synth. The **transposer** solves this problem by keeping a list of the note-ons and their transpositions in an object called **funbuff**, then looking up the transposition when the note-off is played.

Tutorial 27

This patch transposes notes, and makes sure that note-offs get the same transposition as note-ons, even if the interval of transposition changes while the note is being played.


Pitch
Velocity
Transposition
Only note-ons get through the "transpose and store" gate.
gate
+ 0
funbuff
Note-on: untransposed pitch and transposed pitch are stored as an x, y pair. Note-on and note-off: the transposed pitch is sent out, triggered by the untransposed pitch in the left inlet of funbuff.
Transposed Pitch
Velocity

funbuff

An *array* is an indexed list of numbers. Each number in the list has a unique index number or *address*. We'll call the address *x*, and the value stored at that address *y*. The **funbuff** object stores an array of numbers as *x*,*y* pairs.

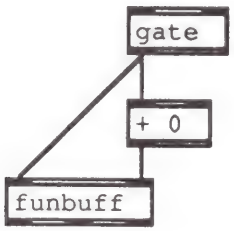
When a number is received in the right inlet followed by a number in the left inlet, the number in the right inlet (*y*) is stored at the address specified in the left inlet (*x*). Then, when an address number is received by itself in the left inlet (*x*), **funbuff** sends the corresponding *y* value out its left outlet.

The numbers can also be stored in **funbuff** as a list: and *x* address and a *y* value. For more information, look under **funbuff** in the *Objects* section of the manual.

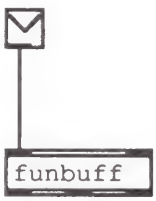
Storing and Recalling Transpositions

The **gate** object in **transposer** is used to pass only the pitch of *note-on* messages. Before the pitch reaches the right inlet of **gate**, the velocity value goes to the control inlet of **gate** and either opens it or closes it. If the velocity is 0 (note-off) the **gate** will be closed and the pitch will not get through.

The note-on pitch goes first to the + object to be transposed, then to the right inlet of **funbuff**. The untransposed pitch then goes to the left inlet of **funbuff**, so the transposed pitch is stored as the *y* value, with the untransposed pitch as the address (*x*).



As soon as the *x*,*y* pair is stored, the untransposed pitch (*x*) is sent by itself, causing the transposed pitch (*y*) to be sent out.



When the note-off message comes later, nothing goes through the **gate**, and the untransposed pitch is sent by itself to **funbuff**, causing the transposed pitch to be sent out again. Since the note-off messages get their transposition from **funbuff** rather than from the + object, the value in the + object can change without affecting the note-off transpositions.

- Close the subpatch window. Play on your MIDI keyboard and drag on the slider at the same time to change the transposition of what you are playing.

Storing transpositions in this manner is essential whenever the interval of transposition is to be changed *while* the notes are being transposed. For example, the transposition might be changed automatically by numbers generated in some other part of the patch.

Differences Between the patcher Object and Your Object

What are the differences between a subpatch in a **patcher** object and a subpatch you created earlier and saved in a separate file?

One difference is in the way they are saved. The subpatch in a **patcher** object is saved as part of the file that contains the **patcher** object. As a result of this, you can edit a **patcher** object subpatch just by double clicking on the **patcher** object and unlocking the subpatch window. When the subpatch is saved as a separate file, however, you can see its contents by double-clicking on the object, but you can't edit the contents of the subpatch window. (Max will not let you unlock it.) To edit the object, you have to open the separate file in which it was created.

The separate file containing your object must be in a folder where the patch that uses it can find it. Max looks for files in a) the same folder as the Max application, b) the same folder as the patch that is using the subpatch, and c) any other folder you have specified in the **File Preferences...** dialog, under *Look for Patcher files and tables in:*. For more information about **File Preferences...**, look under *Menus* in the *Topics* section of the manual.

The other main difference is that if you save your patch while the subpatch window of the **patcher** object is open, it will be opened automatically each time you open the main patch. This is not true of a subpatch that is saved as a separate file.

Beware of Recursion

A patch that is used as an object in another patch can itself contain subpatches. For example, our **transposer** object could have been written to contain a subpatch object called **splitnote** which separated note-on messages from note-off messages.

A subpatch object may not contain itself, however, since this would put Max into an endless loop of trying to load a patch within itself *ad infinitum*. For example, our **transposer** object could not itself contain a **transposer** object, or any subpatch that contains a **transposer** object.

Documenting Your Object

You can see that the **transposer** object has been copiously commented, and all of its inlets and outlets have been given Assistance messages. Such thorough documentation makes it more likely that others will understand your patch and be able to use it, and also helps to remind *you* how your patch works.

Summary

Any patch you create and save can be used as an object in another patch. When you are making a patch that will be used as a subpatch in another patch, you will usually want to include **inlets** and **outlets** (or **sends** and **receives**) so that you can send messages to your object and it can send messages out.

The **funbuff** object stores an *array* of numbers: *x,y* pairs of *addresses* and *values*. When an address number (*x*) is received in the left inlet, the value stored at that address (*y*) is sent out the left outlet. This type of array is useful as a lookup table, for storing values in an indexed list and looking them up later. One use of arrays is to pair note-on pitches with their transposition so that the transposition can be looked up again when the corresponding note-off is played.

The window of a subpatch object which is saved as a separate file is not opened automatically when the Patcher window that contains it is opened (unlike the **patcher** object). An object used as a subpatch can be edited only by opening the file in which it is saved.

Explanatory notes in the form of **comments** and Assistance messages are helpful to you and to others who may use your patch.

See Also

Encapsulation	How much should a patch do?
funbuff	Store x,y pairs of numbers together

Tutorial 28

Your argument

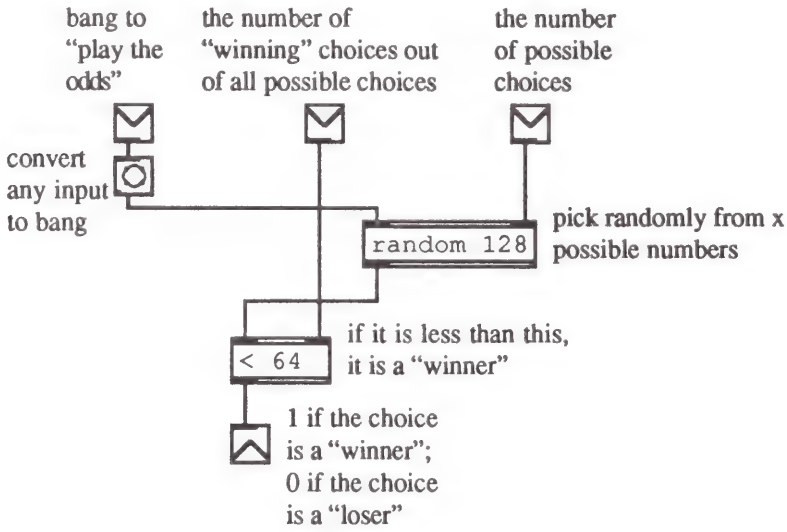
Supplying Initial Values to Your Object

Many Max objects take *arguments*, typed in after the object name, to supply some information to the object such as a starting value. There is a way you can design your own object to get information from typed-in arguments, as well.

The gamble Object

In the Patcher window you can see several instances of an object called **gamble**. It's not a Max object; it's a patch we created and saved in a document named *gamble*.

- Double-click on the **gamble** 64 128 object in the right part of the Patcher window, to see the contents of the subpatch.


bang to "play the odds"
the number of "winning" choices out of all possible choices
the number of possible choices
convert any input to bang
random 128
pick randomly from x possible numbers
< 64
if it is less than this, it is a "winner"
1 if the choice is a "winner"; 0 if the choice is a "loser"

The **gamble** object is an electronic gaming table. When it receives a bang in its left inlet (or anything else, since the **button** inside **gamble** converts all incoming messages to bang), it chooses a random number (limited by the 2nd argument or the number received in the right inlet). If the random number is less than a certain other number (specified by the 1st argument or received in the middle inlet), **gamble** sends out 1. Otherwise **gamble** sends out 0.

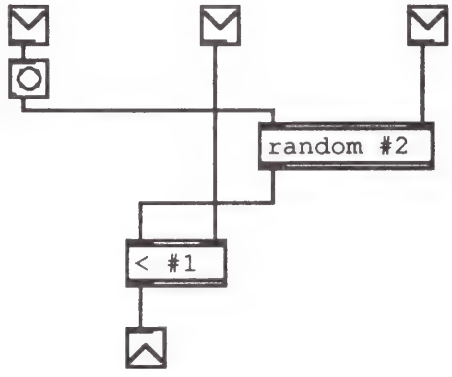
In effect, the arguments to **gamble** state the odds of a 1 being output each time a bang is received in the left inlet. In this case the odds are 64 in 128 (even up).

- Close the subpatch window, and double-click on the **gamble** 2 5 object to see the contents of the subpatch. The odds are different in this subpatch, because the arguments are different.

The # argument

- Now open the document named *gamble* in the Max Tutorial folder. You can see that in the original *gamble* patch, the odds are specified with *changeable arguments*.

Tutorial 28



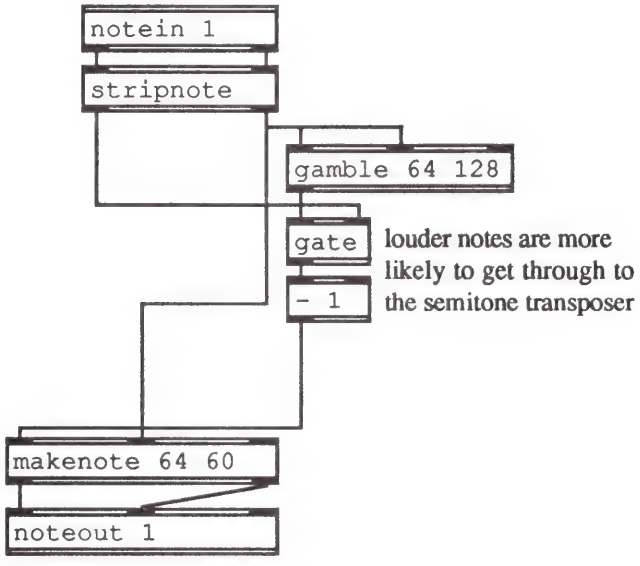
When the *gamble* patch is used as a subpatch in other patches, the changeable **#1** and **#2** arguments are replaced by the 1st and 2nd arguments typed into the **gamble** object. If no argument is typed in, the # arguments are replaced by 0.

The # argument can be used with most Max objects inside your object, and can be replaced by a symbol as well as a number. For examples of its usage, look under *Arguments* in the *Topics* section.

Using Weighted Randomness

Now that you have seen how arguments are used to set initial values for a subpatch object, let's see how **gamble** is actually used in this patch. Each time **gamble** receives a bang in its left inlet, it makes a *probabilistic decision* whether to send out 1 or 0, depending on the specified odds.

In the right portion of the Patcher window **gamble** is used to decide whether to open or shut a **gate**.



Tutorial 28

Your argument

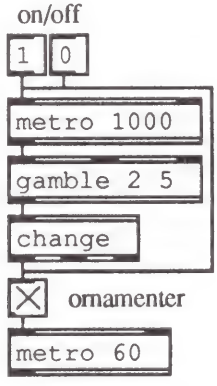
The velocity of each played note sets the odds of the **gate** being opened, then **gamble** is triggered to open or shut the **gate** based on those odds. If the **gate** is open, the pitch will get through and will be transposed down a semitone and transmitted back to the synth.

Let's say you play a note with a velocity of 93. The odds of the gate being open are 93 in 128, a little less than 3 in 4, so it is likely that the note you play will be transposed. If you play a note with velocity of 3, however, the odds are only 3 in 128 of the **gate** being open, so the note will probably not be transposed.

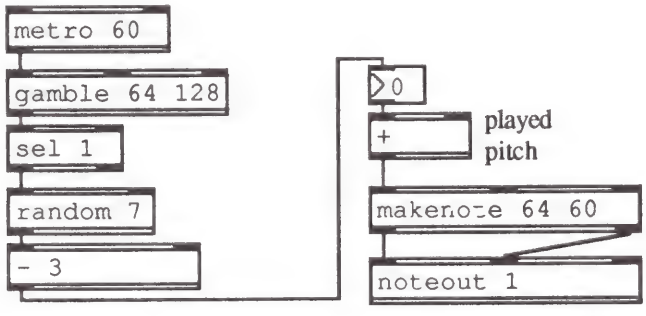
The result is a probabilistic "Thelonius Monk effect" of adding lower grace notes to more and more pitches as the velocity increases. Notice that we don't need to use the **transposer** object shown in the previous chapter, because we are transposing only note-on pitches and **makenote** provides the note-offs.

- Play with different extreme velocities on your MIDI keyboard and notice the difference in likelihood of a grace note being added to what you play.

In the left side of the Patcher window **gamble** is again used to make *weighted random decisions*, with two slightly different implementations. When the **metro** 1000 object is turned on, it triggers **gamble** every second, and **gamble** turns the **metro** 60 object on or off (it will be turned on approximately 40% of the time).



Every 60ms **gamble** will send out either a 1 or a 0, with the odds depending again on the velocity of the played note. When it is 1, **sel** triggers **random** to choose a random ornamentation interval which is added to the played note and transmitted to the synth.



A little bit of additional calculation is performed to make the range of the ornamentation interval also depend on the played velocity. When the velocity is at a maximum, the range of the ornamentation will vary from -7 to 7 semitones (up or down as much as a perfect fifth). When the velocity is at a minimum, the ornamentation will only be 0 (unison).

For example, when the velocity is 127, a random number is chosen between 0 and ((127+8)÷9)-1, that is, 14. That number will then have ((127+8)÷9)÷2) subtracted from it, i.e. 7, setting the range of possible ornamentations from -7 to 7.

- Turn the **metro** 1000 object on, and play on your MIDI keyboard with extreme changes of pitch and velocity. Notice that the ornamentation is wider and more dense when you play harder. The effects of the ornamenter are most comprehensible when you play very sparsely on the keyboard.

When to Use Arguments

The reason for supplying values to an object is to modify some characteristic of the subpatch. If you always want the subpatch to do exactly the same thing, you probably don't need to change the values inside it in any way. If, however, you want your object to do something slightly differently depending on some value it receives, the value will have to be supplied using an inlet or a typed-in argument.

There's no hard and fast rule about when to supply values to a subpatch by using arguments, and when to supply values via inlets. Generally speaking, if you will just want to supply the value once it can be most easily given as an argument, but if you want to change the value of a single object often you will need to use an inlet.

One solution is to make both ways possible, as we have done with **gamble**. The arguments are used to set initial values inside the subpatch, but the values can be changed by numbers received in the middle and right outlets.

Summary

You can enable your object to accept information from typed-in arguments by including changeable # arguments in the subpatch. A changeable argument of #1 in the subpatch is replaced by the first typed-in argument in the object box, #2 is replaced by the second argument, and so on. If no argument is typed into the object box, the changeable argument is set to 0.

Your patch can make *weighted random* (*probabilistic*) decisions by choosing a random number, then testing to see if the number meets certain conditions.

See Also

Arguments	$ and #, changeable arguments to objects

Tutorial 29

Test 5: Probability object

Create Your Own Object

This is an exercise in the creation and use of your own object. We'll make an object that passes on a certain percentage of the bangs it receives, then use that object in a patch. First we must create the object.

1. Create an object called **MyPassPct** that receives bangs in its left inlet and passes a certain percentage of them out its outlet. The percentage should be specified by a typed-in argument or by a number received in the right inlet.

Hints

A *percentage* of probability is the number of times an event is likely to occur in 100 tries. For example, a 33% chance means the odds are 33 in 100 that the event will occur.

Use the **gamble** object from the previous chapter as a model to give you an idea how to proceed. The **MyPassPct** object will be similar except:

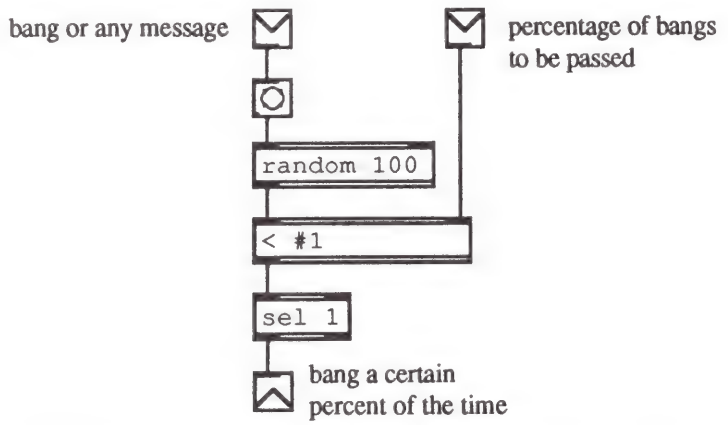
a) The number of possible random choices will always be 100.

b) Instead of sending out a 1 or a 0, you want your object to send out bang (whenever the condition is met).

Solution to Exercise 1

We have saved our solution to Exercise 1 in a file called *PassPct* in the Max Tutorial folder.

When a bang or any other message is received in the left inlet, the **random** 100 object chooses a number from 0 to 99. If it is less than the number specified as an argument (or received in the right inlet), **sel** sends a bang out the outlet.



Next we will use the **PassPct** object in a patch to make probabilistic decisions.

Tutorial 29

Pass a Percentage of bangs from a metro

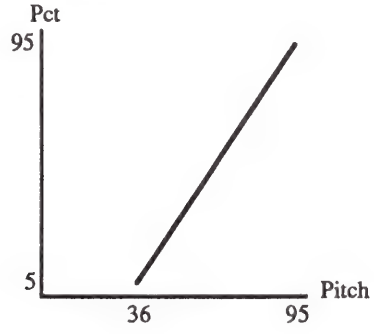
2. Use a **metro** to send bangs at a constant speed and use **PassPct** to pass only a certain percentage of those bangs. Use the bangs passed by **PassPct** to trigger notes sent to the synth. Use a 5-octave **kslider** to choose which pitch will be transmitted.

   Make the percentage value of **PassPct** depend directly on the pitch selected with **kslider**. As the pitch increases from 36 to 95, the percentage should increase from 5 to 95.

Hints

The pitch value sent out by **kslider** should be stored in some type of storage object (an **int**, a **value**, a **Number box**, etc.—an **int** is the most efficient). The bangs from **PassPct** can then trigger the storage object to send its number to **makenote** and play a note.

The hard part of this exercise is using the range of pitches sent out by **kslider** (from 36 to 95) to provide a different range of percentages (from 5 to 95) to **PassPct**. This is known as *mapping* one range to another. A direct correspondence such as this is a *linear* map: the relationship between the two ranges can be graphed as a straight line.



As the pitch changes from 36 to 95, the percentage changes from 5 to 95

Calculating a Linear Map

The problem of linear mapping is: given one range of numbers from *xmin* to *xmax* and another range of numbers from *ymin* to *ymax*, and given some number *x* within the first range, find the number *y* that occupies the same position in the second range.

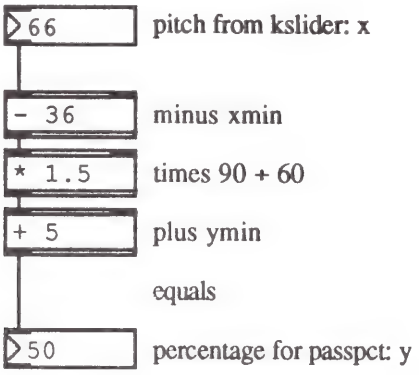
Here's a formula for finding the *y* value that corresponds to any given *x* value.

$$y = ((x - xmin) * (ymax - ymin) \div (xmax - xmin)) + ymin$$

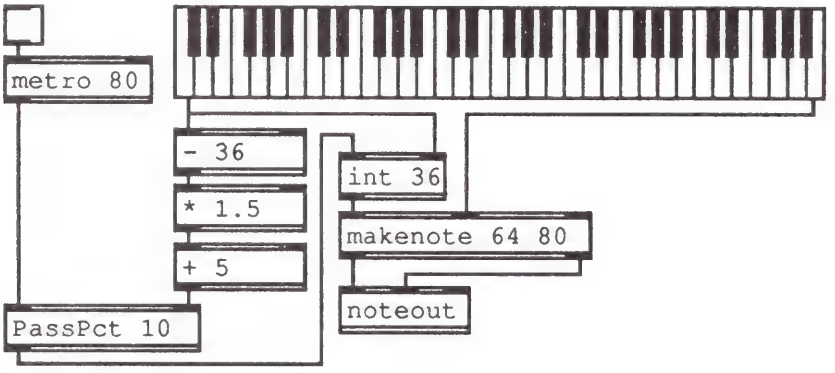
When we plug our ranges into the formula, we get

$$y = ((x - 36) * 90 \div 60) + 5$$

How do we translate this into objects?



The pitch from **kslider** is sent into the formula, and the percentage is sent to **PassPct**. Your patch might look something like this.



Use PassPct to make Random Choices

Let's add one more element to the exercise.

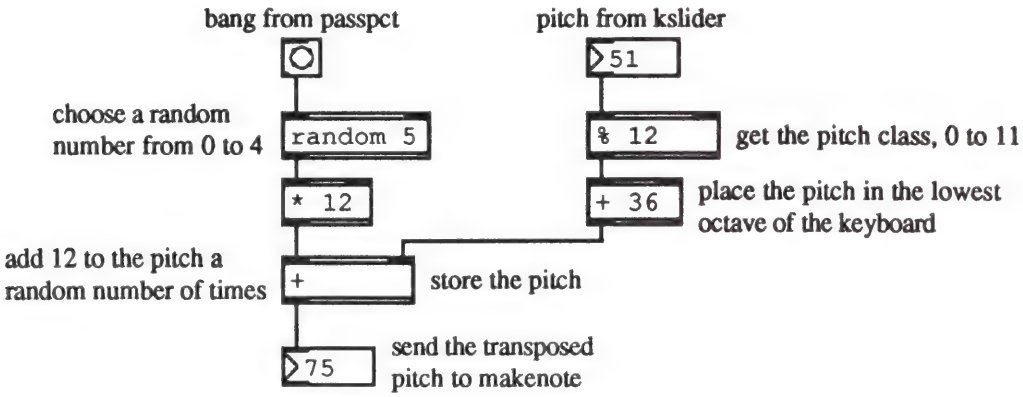
3. Add another **PassPct** object that receives bangs from the same **metro**, but triggers random octave transpositions of the selected pitch. Make the percentage of this **PassPct** object *decrease* from 95 to 5 as the selected pitch increases.

This part of the exercise presents two new problems: how to create random octave transpositions of a pitch, and how to express an inverse linear relationship. Try to find a solution to these problems yourself before reading further.

Random Octave Transpositions of the Pitch

To make a random octave transposition of a note, you need to calculate the *pitch class* of the note (C, C#, D, etc.), then add 12 to that pitch class some random number of times. You will want to

limit the random numbers so that they keep the transpositions within the range of the keyboard. The solution might look something like this:


bang from passpct
pitch from kslider
51
choose a random number from 0 to 4
random 5
% 12
get the pitch class, 0 to 11
* 12
+ 36
place the pitch in the lowest octave of the keyboard
add 12 to the pitch a random number of times
+
store the pitch
75
send the transposed pitch to makenote

Calculating an Inverse Linear Map

You may remember from an earlier patch in which we inverted pitches that we subtracted the maximum possible pitch from whatever pitch was played, then took the absolute value of the result. (See Patch 2 in Tutorial 14.)

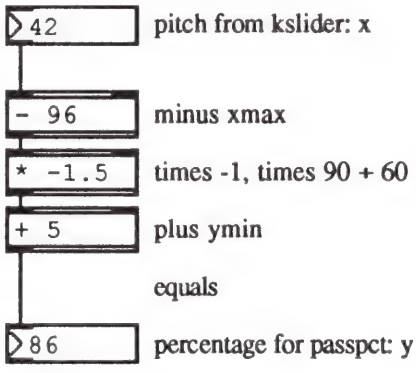
The formula for an *inverse* linear map, then, looks like this:

$$y = (-(x - xmax) * (ymax - ymin) + (xmax - xmin)) + ymin$$

When we plug our ranges into the formula, we get

$$y = (-(x - 96) * 90 + 60) + 5$$

We can translate this into Patcher as



Scroll to the right in the Patcher window to see Exercise 2 and Exercise 3 combined in a single patch. (We've used our **PassPct** object).

Tutorial 29

Test 5: Probability object

Summary

To create your own object, make a patch that includes **inlet** and **outlet** objects (and changeable **#** arguments if appropriate), save the patch, then use your object by typing the name of the file into an object box in some other patch.

The **PassPct** object is similar to the **gamble** object from the previous chapter. It passes or suppresses the bangs it receives, according to some percentage of probability.

You can transpose a note by an arbitrary number of octaves by first calculating its *pitch class* (with a **%** 12 object), and then adding some multiple of 12 to the pitch class.

You can create a direct or inverse linear relationship between two ranges of numbers using the *linear mapping* procedure described in this chapter.

Number groups

Tutorial 30

Use of Lists

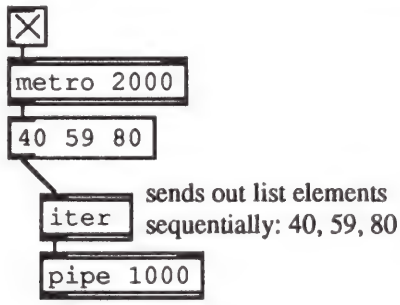
We have seen that a message can consist of a single number or a *list* of numbers separated by spaces. The list is an effective way of sending numbers together, ensuring that they are received at the same time by an object.

For example, we usually want to keep pitch values and velocity values synchronized so that they are received in the proper order by **noteout**. When **noteout** receives a list in its left inlet, it interprets the third element (if present) as the channel number, the second element as the velocity, and the first element as the pitch.

There are objects specifically for combining numbers into a list, and objects for breaking lists up into individual numbers. So, you can choose the most appropriate way to send groups of numbers between objects. A list even can include symbols (words) as well as numbers, which may be useful in some cases. As long as the first element is a number, Max objects will recognize the message as a *list*.

iter

When the **iter** object receives a list of numbers in its inlet, it breaks the list up into its individual elements and sends the numbers out in sequential order rather than all at the same time. It's as if **iter** puts commas between the elements, to make them into separate messages.



In the right part of the Patcher window you can see **iter** at work. When the **metro** triggers the list of numbers, it is sent to **iter**, which breaks up the list and sends each of the numbers on in order, as rapidly as possible. The numbers are delayed by the **pipe**, then are sent on as (virtually simultaneous) pitches to **makenote**.

unpack

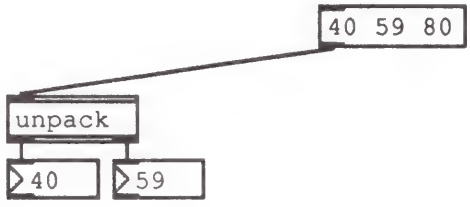
When a list is received by **unpack**, each element of the list is sent out a different outlet. The number of outlets **unpack** has is determined by the number of arguments you type in. (The arguments also set an initial value for each outlet.) If there are no typed-in arguments, **unpack** has two outlets, both with an initial value of 0.

Tutorial 30

Number groups

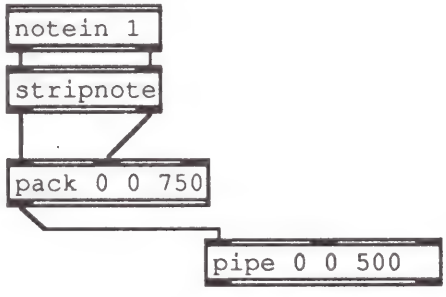
If there are more elements in the incoming list than **unpack** has outlets for, the extra numbers are ignored. The numbers that are received by **unpack** are also stored, so if a list is received that has *fewer* numbers than there are outlets, **unpack** sends out the stored number for the extra outlets.

In the example patch, when a list is received by **unpack**, the second item in the list is sent out the right outlet, then the first item in the list is sent out the left outlet (output order is always right to left).



pack

The **pack** object combines separate numbers into a list. It stores the number most recently received in each of its inlets, then when it receives a number or a bang in the *left* inlet it sends out all the stored numbers together as a list. The number of inlets—and the initial value stored in each one—is specified by the typed-in arguments.



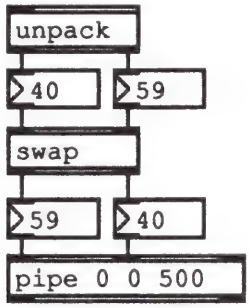
In the left part of the Patcher window, note-on pitch and velocity values from your MIDI keyboard are packed in a list along with the number 750, and the list of pitch-velocity-delay is sent to the **pipe**. Every note from the keyboard will be delayed 750ms, even if the delay time of the **pipe** is changed by some other part of the patch, because the delay time is sent in the same list as the note-on data.

swap

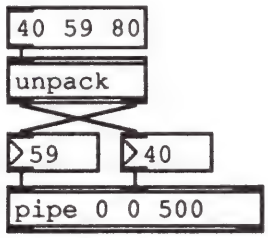
The **swap** object reverses the sequential order of numbers it receives. It is triggered by a number in its left inlet, just like other objects, but it sends that number out its right outlet *first*, then sends the number that was received earlier in the *right* inlet out its left outlet.

Tutorial 30

In the example patch, **swap** reverses the order of the first two list items, received from **unpack**, and uses the first number in the list, 40, as a velocity and the second number, 59, as a pitch.



It would not be sufficient just to cross the patch cords from **unpack**, because the number 59 would arrive at the left inlet of **pipe** and trigger it before the number 40 got there.

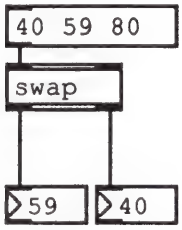


This patch is not equivalent to the one shown above.

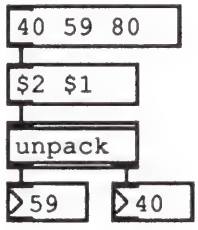
When **swap** receives a **bang** in its left inlet it sends out the same numbers again. The two numbers to be swapped can also be received in the left inlet as a list. In fact, the **unpack** object in this patch is not strictly necessary, because **swap** would understand the list and swap the first two items, but we included **unpack** to make the number-swapping more evident visually. There is also an object for swapping floats, called **fswap**, not demonstrated here.

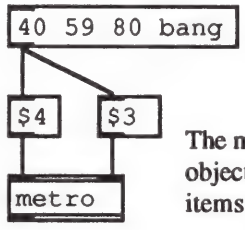
Lists Can Be Managed with Message Boxes

As was shown in Tutrorial 25, a message box can also be used to isolate and rearrange items in a list. Here are a couple of examples showing possible uses of message boxes for selecting individual items from a list.



These two patches are equivalent, although the first way is more efficient.





The message object can isolate items in a list.

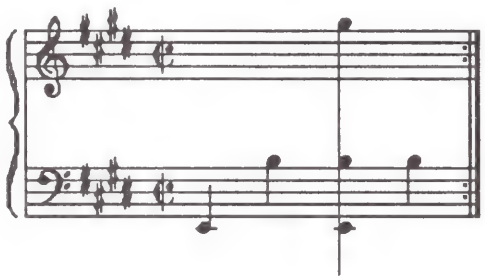
Tutorial 30

Number groups

An Automatic Accompanist

Now that we have seen how the list management objects work, let's see how they are used in the example patch. Elements of the list 40 59 80 are rearranged and delayed in different ways to send different messages to **makenote** at different times.

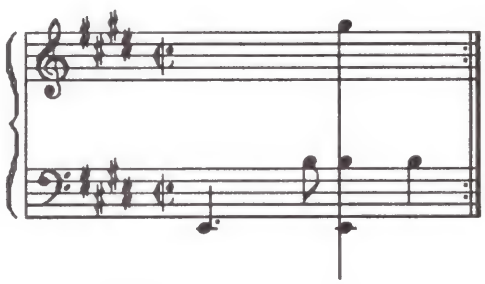
When the **metro** is turned on, the entire list is sent to **makenote** immediately, playing the note 40 (E1) with a velocity of 59 and a duration of 80ms. The pitch and velocity are reversed by **swap**, and delayed 500ms before being sent to **makenote**, playing the note 59 (B2) with a velocity of 40. One second after the **metro** was turned on, the numbers are all sent to **makenote** as a chord—E1, B2, and G#4—with the velocity of 40 from the previous note. At the same time, the bang that was delayed by the **del** object retriggers the note B2 from swap, and it is delayed another 500ms before being played. After a total of 2 seconds, the entire process is repeated. The result sounds like this:



Automatic E-major accompaniment figure

Note-on pitches and velocities from your MIDI keyboard are packed into a list along with a delay time and sent to **pipe** with a delay of 750ms. This causes a short-note echo of every played note 750ms later.

The played notes also have an effect on the accompaniment. If a played note arrives at **pipe** in between the first and second notes of the accompaniment figure, the delay of the second note of the accompaniment will be 750ms, causing this rhythmic change:



Also, if a delayed played note reaches **makenote** between the second and third notes of the accompaniment, the velocity of the chord will be altered.

- Turn on the accompaniment and play a melody along with it.

Number groups

Tutorial 30

Summary

A *list* is any message that begins with a number and contains additional items as *arguments*. Usually the arguments are all numbers, but they may also be symbols.

Sending numbers together as a list ensures that they will be received together. Many objects, such as **pipe**, **makenote**, and **noteout**, interpret a list of numbers received in the left inlet as if the numbers had been received separately in different inlets.

The **pack** object combines the numbers it receives in each inlet into a single list of numbers. The **unpack** object breaks a list of numbers up into its individual *elements* (items), and sends each element out a different outlet, in order from right to left. The **iter** object sends each element of a list individually, in order from left to right, out a single outlet.

The changeable **$** argument in a message box can be used to isolate individual elements of a list. This is especially effective if the list contains symbols in addition to numbers.

The **swap** object reverses the sequential order of two numbers. When a number is received in the left inlet, it is sent out the right outlet, then the number that was received *earlier* in the right inlet is sent out the left outlet.

See Also

buddy	Synchronize arriving numbers, output them together
fswap	Reverse the sequential order of two decimal numbers
iter	Break a list up into a series of numbers
pack	Combine numbers into a list
swap	Reverse the sequential order of two numbers
thresh	Combine numbers into a list, when received close together
unpack	Break a list up into individual numbers

Tutorial 31

Using timers

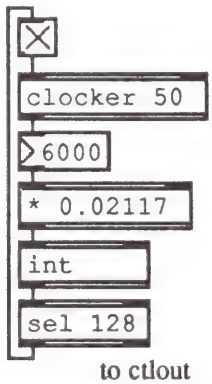
Timed Processes

So far we have used two different timing objects: **metro** for sending a bang at regular intervals, and **timer** for reporting the elapsed time between two events. In this chapter we introduce some objects for producing timed progessions of numbers.

clocker

The **clocker** object is the same as **metro**, except that instead of sending out bang at regular intervals it sends out the time elapsed since it was turned on. With this information you can cause values to change in some manner correlated with the passing of time.

In the part of the Patcher window marked A, a **clocker** reports the elapsed time, and that information is mapped to send increasing values to the mod wheel of the synth. Over the course of 6 seconds the time progresses from 0 to 6000, causing the control values to increase from 0 to 127. When the value reaches 128, the **clocker** is turned off by **sel**. The result is a 6-second linear *fade-in* of the modulation effect on the synth.



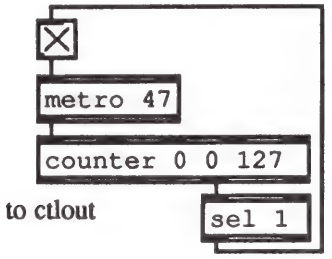
The **int** object is included to truncate the float output of the * object so that **sel** will make an accurate comparison.

counter

The **counter** is not itself a timing object, but it is frequently used in conjunction with **metro**, because **counter** counts the number of bangs it has received. The **metro-counter** combination is an effective way to increment or decrement a value repeatedly.

In the part of the Patcher window marked B, the first argument to **counter** specifies the direction of the count: 0 for upward. (1 is for downward, and 2 is to go back and forth between up and down.) The second argument sets the minimum value of the count, and the third argument sets the maximum value.

Tutorial 31



Note: The meaning of the arguments to **counter** changes depending on how many arguments there are. Look under **counter** in the *Objects* section for details.

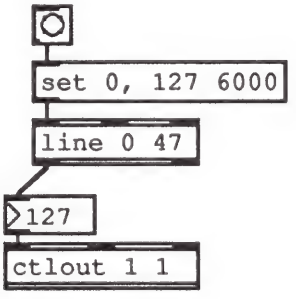
The count is sent out the left outlet. When the maximum (127) is reached, **counter** sends a 1 out its right-middle outlet. This 1 is detected by **sel**, which toggles the **metro** off. This is another way to get the same effect as we did using **clocker**. With **counter**, however, the numbers can be easily placed in the desired range (0 to 127 in this case) without a multiplication being performed each time. Multiplication takes longer for a computer to perform than addition.

The **metro** is set to a speed of 47ms so that the progression from 0 to 127 will be completed in 5.969 seconds—as close as possible to 6 seconds (using this method).

line

The **line** object also outputs numbers in a linear ramp from some starting value to some ending value over a specific period of time. The first argument sets the starting value and the second argument sets the *grain*—the time interval at which numbers will be sent out. When a time period is received in its middle inlet and an ending value is received in its left inlet, **line** outputs numbers in a linear progression from the starting value to the ending value over the specified time period.

The numbers in the inlets can also be received together as a *list* in the left inlet. If a number is received by itself in the left inlet, without a time period being received at the same time, **line** jumps to (and outputs) the new value immediately. A starting value can be sent to **line** without triggering any output by sending it a set message (the word set, followed by a number).



Set starting value to 0, then progress to 127 in 6 seconds, outputting a number every 47ms

Tutorial 31

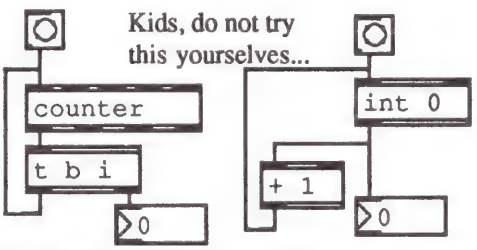
Using timers

Stack Overflow

Have you ever been in the position of feeling like the list of things you have to do is growing faster than you can get them done? Well, it's possible to overload Max in a similar way, so that the list of things Max has to do eventually overflows the amount of memory space available for its *stack* of things to do. This is known as a *stack overflow*, and it causes Max to shut down its internal scheduler and stop performing timed operations until you fix whatever is causing the overflow.

One way to cause a stack overflow is to feed an object's output back into its input. For example, when you want to increment numbers as fast as possible, you might be tempted to feed the output of an object like **counter** right back into itself, repeatedy incrementing the count. But such automatic repetitions must be separated by at least a millisecond or two, otherwise Max will generate repetitions too fast for itself to keep track of, and you will get a *Stack Overflow* error dialog. When this happens, you must choose **Resume** from the Max menu to restart Max's scheduler.

- Patches D and E show two examples of situations that result in stack overflow. Click on the buttons if you want to make Max very unhappy. (Go ahead, you won't break anything.) Remember to choose the **Resume** command to start Max up again.



These may look like good ways to send out numbers as fast as possible, but they will result in stack overflow.

tempo

The **tempo** object is another metronome, but it operates in somewhat more traditional musical terms than the millisecond specifications necessary with **clocker**, **metro**, and **line**. The first argument to **tempo** (or a number received in the left-middle inlet) sets a metronomic tempo in terms of *beats per minute*—that is, quarter notes per minute—just like a traditional metronome.

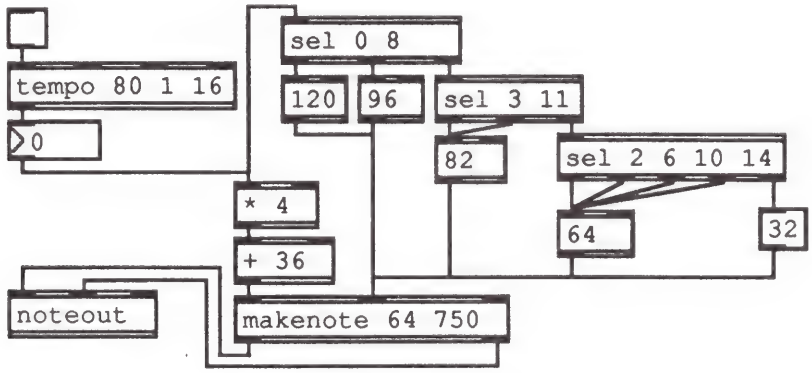
The second and third arguments (or numbers supplied in the right-middle and right inlets) specify what fraction of a whole note **tempo** will use to send out ticks of the metronome. For example, if the second and third arguments are 1 and 16, the fraction is $1/16$ of a whole note and **tempo** sends out a number from 0 to 15 for every sixteenth note, based on the specified quarter note tempo. A fraction of $2/3$ would send out half note triplet ticks (a tick every $2/3$ of a whole note), and so on.

The numbers sent out by **tempo** always go from 0 to the number 1 less than the pulse division (the third argument). The greatest allowable division is 96 (sixty-fourth note triplets). The fact that

Tutorial 31

tempo sends out a number (a sort of pulse index), lets you assign different things to happen on different pulses in a measure. In this way you can generate metrically-based automated processes.

In Patch F, **tempo** sends out a number for each sixteenth note at a tempo of 80, and triggers a different pitch and velocity for each pulse of the measure. The pitch ascends in an arpeggiated augmented triad, and the velocities are greater on the strong beats of the 4/4 measure, and smaller on the weaker pulses.



- Select a velocity-sensitive sound on your synth and turn on the **tempo** object.

External Timing

The **metro**, **clocker**, and **tempo** objects can be synchronized to some timing source other than Max's internal millisecond timer, such as a time-code generator, an external sequencer, or even some other software sequencer such as Opcode's *Vision*. For details, look under those object names in the *Objects* section.

Summary

When **clocker** is turned on it sends out the elapsed time at regular intervals. The time value can be mapped to other ranges to to make them depend on the passing of time.

The **counter** keeps track of how many bangs it has received and sends out the count. The count can be restricted to a specific range, and the bangs can be supplied repeatedly by a **metro** to increment and/or decrement the **counter** at a specific speed. This is another way of creating a particular progression of numbers over time.

The **line** object is a third way of generating a linear progression of numbers. **line** outputs numbers in a ramp from some starting value to some ending value, arriving at the new value in a specific amount of time.

Incrementing numbers by means of recursive loops, without some type of delay between repetitions, can result in a *stack overflow* error, which causes Max to stop its internal scheduler. Choosing **Resume** from the Max menu restarts the scheduler.

Tutorial 31

Using timers

The **tempo** object is a metronome that lets you specify timing in traditional musical terms of *beats per minute* and beat divisions. It sends out a different number for every pulse in a measure, so each pulse number can trigger a different action.

If you are using the MIDI Manager, the **metro**, **clocker**, and **tempo** objects can be synchronized to an external timing source such as a sequencer or a time-code generator.

See Also

clocker	Output the elapsed time, at regular intervals
counter	Count the bangs received, output the count
line	Output numbers in a ramp from one value to another
metro	Send a bang, at regular intervals
timein	Report time from external time code source
Ports	How ports are specified
tempo	Output numbers at a metronomic tempo

The **table** *object*

Tutorial 32

An Indexed Array of Numbers

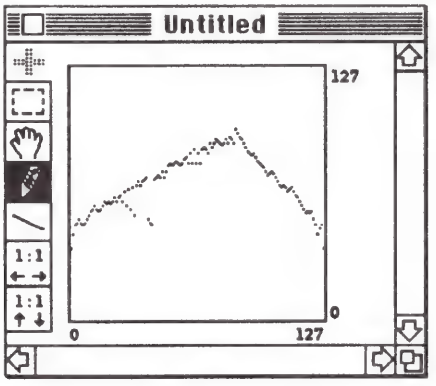
In Tutorial 27 we introduced the **funbuff** object for storing an indexed array of numbers. Number values are stored with an index number (address), then when you want to recall a value you just specify the address where it is stored. The **table** object stores and recalls numbers very similarly, but has many more features.

Graphic Editing

The most notable feature of the **table** object is that it allows you view and edit the stored numbers in a graphic editing window.

- Check **All Windows Active** in the Options menu so that you can view **tables** and click in the Patcher window at the same time.

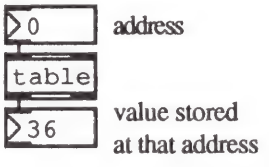
- Double-click on the **table** object at the bottom of Patch 1. A Table window will open to show a graph of some numbers that we have stored there.



This **table** contains 128 numbers, with addresses from 0 to 127. Addresses always go from 0 to the number 1 less than the size of the **table**. This **table** shows a range of possible *values* from 0 to 127, and the values we have stored range from 36 to 96.

- Turn on the **metro** at the top of Patch 1. The **counter** counts up and down between 0 and 127. The numbers are sent through a **uslider** just to show their progression graphically, then they are sent to the left inlet of **table**.

A number by itself in the left inlet of **table** specifies an address, and the value stored at that address is sent out the left outlet. The output of the **table** is displayed graphically by the second **uslider**. You can see and hear the numbers in the **table** as **counter** steps through them.



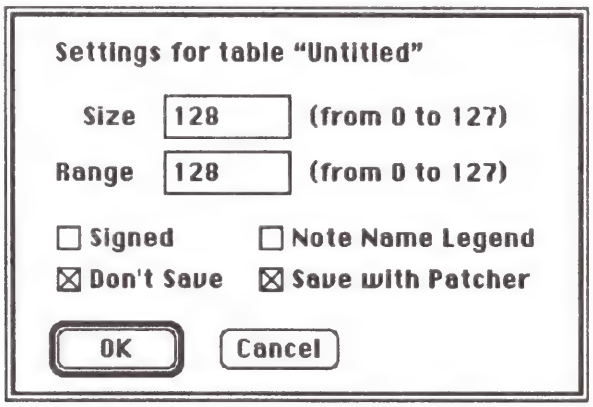
Tutorial 32

The **table** *object*

Get Info...

- With the *Untitled* table editing window still in the foreground, choose **Get Info...** from the Max menu.

The **Get Info...** dialog shows you the *Size* of the **table** (the number of storage addresses) and the *Range* of displayed values. It also has two options for viewing the numbers. Checking *Signed* causes the Table window to display negative values as well as positive, and *Note Name Legend* shows the *y* axis values as MIDI note names instead of numbers.



Saving the Values in a table

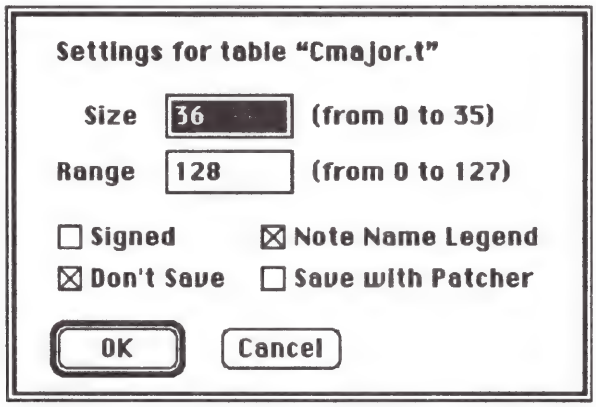
It's important to understand the different options for saving a **table**, so you don't lose numbers you've carefully entered. Normally, the values you store in a **table** object are lost when you close the Patcher window. If you check *Save with Patcher*, however, the numbers in the **table** will be saved as part of your Patcher document. Then, when you reopen the patch the **table** will still contain the numbers. We have checked *Save with Patcher* for this **table** so that our masterpiece will be preserved.

If you change the values in a **table**, Max will ask you if you want to save the changes you made to that **table** when you close the Patcher window. If you don't want Max to ask you that every time you close the Patcher window, check *Don't Save*. *Don't Save* does not cause you to lose any values you have explicitly saved with *Save with Patcher*; it just doesn't remind you to save any subsequent *changes*.

Any time the table editing window is in the foreground you can save its contents to a *separate file* by choosing **Save** from the Max menu. Then, whenever you want to use the file in a patch, just create a **table** object and type in that file name as an argument. The contents of the file will be loaded into the **table** object.

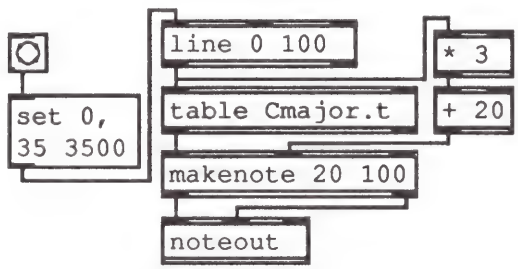
An example of a **table** that's stored as a separate file can be seen in Patch 3. The file *Cmajor.t* is loaded into the **table** object whenever the Patcher window is opened. You might want to give your Table files names that include some distinguishing characteristic, such as *.t*, so that you can tell Table files and Patcher files apart.

- Double-click on the **table** Cmajor.t object to see its contents. With the *Cmajor.t* table editing window in the front, choose **Get Info...** from the Max menu.



You can see that the *Size* of the **table** is 36 (the number of notes in C-major that are on a 61-note keyboard). *Don't Save* is checked because we don't anticipate wanting to save changes to this file, and *Save with Patcher* option is checked because it's sufficient to have the **table** stored in a separate file and read it in when we open the patch.

- Close the **Get Info...** dialog and click on the **button** in Patch 3 to hear the use of **line** and **table** for reading through a predetermined set of pitches. Notice that **line** is also used to create a velocity crescendo from 20 to 125.



Creating a New table

There are three easy ways to create a new table file.

1. Choose **Table** from the New menu to open a new table editing window. Draw in the values you want, then choose **Save** from the File menu to save the table values.
2. Create a new **table** object, which will automatically open a new table editing window for you. The new **table** can be saved as a separate file before closing the table editing window, or you can check *Save with Patcher* so that it will be saved as part of your patch.
3. Choose **Text** from the New menu to open a new Text window. Type in the word table, followed by a list of numbers, then save the file.

Tutorial 32

The **table** *object*

Once you have saved a Table file you can use it in a Patcher window by creating a new **table** object and typing in the file name as an argument.

Drawing in a Table Window

You can *draw* numbers into in a table editing window with the *Pencil tool*, or use the *Straight Line tool* which automatically draws a straight line between the points where you click.

show crosshairs, to aid precision drawing

select a region to cut, copy, clear, or paste

drag the display, to see another part of the table

draw in values freehand with the mouse

draw values in a straight line, from click to click

zoom the horizontal display in or out

zoom the vertical display in or out

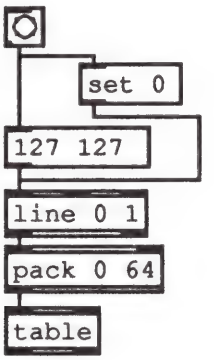
You can select a *region* of values with the *Selection tool*, cut or copy the values, then select another region and paste the first region in its place. You can even copy numbers from a Text window—or from any word processing application—then paste them into the graphic table window in a region specified with the *Selection tool*.

- Double-click on the **table** object at the bottom of Patch 1 and draw a new melodic curve, then listen to it by turning on the **metro**.

Other Ways to Alter a table

The **table** object can understand a number of messages in its left inlet. For a complete list, look under **table** in the *Objects* section. Patch 2 shows a few of the different messages, showing how you to alter a **table** without opening its graphic editing window.

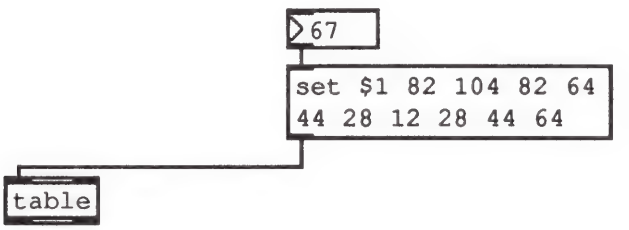
To store values in a **table**, send the value in the right inlet, then send the address where you want it stored into the left inlet. You can also send the address and the value to the left inlet together as a list. In Patch 2, we use a **line** object to send lists to **table** automatically, filling all its addresses with the value 64.



- Double-click on the **table** object in Patch 2, then click on the **button** to see the value 64 being stored at all the addresses.

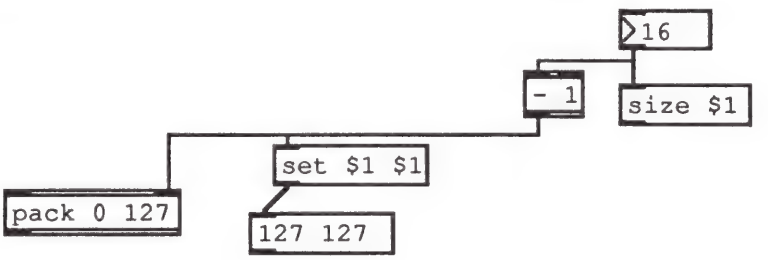
The word set, followed by an address and one or more values, stores the values starting at the specified address. For example, the message set 23 65 68 79 stores the number 65 at address 23, 68 at address 24, and 79 at address 25.

- Send address numbers from the **Number box** to trigger the **message** containing set $1. Watch the results in the table editing window.



- The word size, followed by a number, sets the size of the **table** (the number of addresses). Trigger the message constaining size $1 by sending it a number from the **Number box**. Notice the change in the table editing window.

You'll also notice that we've included additional connections so that a new size setting will cause corresponding changes in other objects, so they interact properly with the **table**.



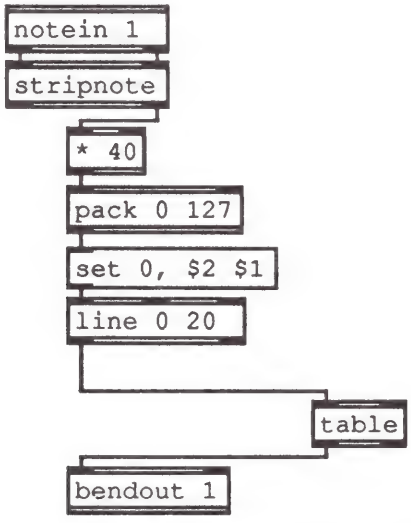
Tutorial 32

The **table** *object*

Using a table for MIDI Values

In Patch 1 we used the values in a **table** to provide pitches to **noteout**. In Patch 2 we use a **line** object to step through the **table** at different speeds, outputting different pitch bend curves.

Each note-on velocity is multiplied by 40 (yielding potential values from 40 to 5080). This value is used as the amount of time the **line** object will take to read through the **table**. The louder a note is played, the more slowly **line** reads through the table, sending out pitch bend values.



Notice how the message box is used to rearrange the incoming numbers and send out two different messages.

- Alter the values in the **table** in the ways discussed, or by drawing in a curve yourself. Play a melody on your MIDI keyboard with long notes and a variety of velocities, to hear the different pitch bend speeds.

Summary

The **table** object stores and recalls an indexed array of numbers. You can graphically view and edit the stored numbers by double-clicking on the **table** object.

The values in a **table** are normally discarded when the Patcher window is closed, but you can save them as part of the patch by selecting the **table**, choosing **Get Info...** from the Max menu, and checking *Save with Patcher*. You can also save a **table** as a separate file, and can then use it in a patch by creating a **table** object and typing in the file name as an argument.

To open a new Table window, choose **Table** from the New menu, or create a new **table** object in a Patcher window. You can also just type the word table, followed by a list of numbers, into a Text window and save it as a file.

The **table** *object*

Tutorial 32

To store values in a **table** object without opening its graphic editing window, send the *value* in the right inlet then send the *address* where you want to store it into the left inlet. Alternatively, you can send the address and value in the left inlet together as a list.

A set message changes certain values in the **table**, and a size message changes the number of values the **table** can hold.

See Also

table	Store and graphically edit an array of numbers
Tables	Using the **table** graphic editing window

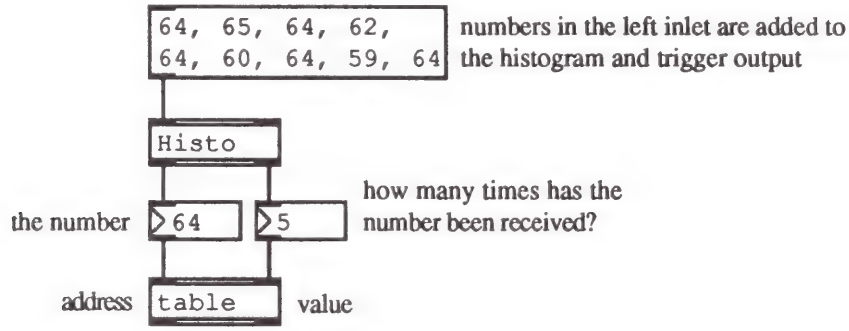
Tutorial 33

Probability tables

Making a Histogram

A *histogram* is a graph of frequency distribution, showing the relative occurrence of different events. The **Histo** object keeps track of all the numbers it receives, as well as how many times it has received each number, in an internal histogram.

Each time **Histo** receives a number between 0 and 127 in its left inlet, it adds that number to its internal histogram, then sends the number of times it has received that number out the right outlet, and the number itself out the left outlet. This output can be sent directly to a **table** to keep a graphic representation of the histogram. The *addresses* in the **table** correspond to the numbers received by **Histo**, and the *values* in the **table** tell the frequency of occurence of each number.



The frequency distribution of different numbers—a comparison showing which numbers occurred most frequently—is displayed in the graphic window of the **table**.

Probability Distribution

The bang message in the left inlet of a **table** has a special function. Instead of sending out a stored value, the **table** sends out an *address*. The probability of a particular address being sent out is in direct proportion to its stored value, as compared to the other values in the table. If the value stored in an address is greater than in other addresses, that address is more likely to be sent out when a bang is received. For a more detailed description of the effect of bang on a **table**, look under *Quantile* in the *Topics* section.

This feature of the **table** makes it perfect for storing a probability distribution. Each address can be assigned a different likelihood of being sent out when a bang is received. If the values in the **table** have been supplied by **Histo**, as described above, the likelihood of a number being sent out of the **table** depends on how many times it was received by **Histo**.

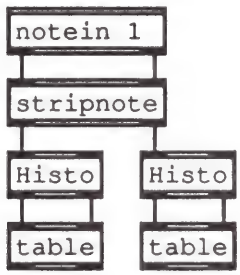
With this combination you can base the probability of a number's occurrence on the past history of how many times it has already occurred. The more it has occurred in the past, the more likely it is to occur in the future.

Probability tables

Tutorial 33

Keeping a History of What is Played

In our example patch, we have used **Histo** and **table** to keep a frequency distibution of the pitches and velocities of notes played on the synth. These **tables** store histograms of the pitches and velocities played.



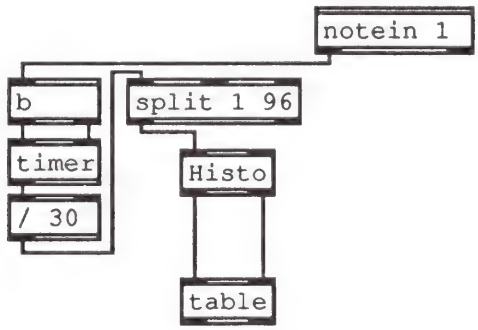
The **stripnote** object is very important here because without it note-off messages would cause each pitch to be counted twice, and the velocity 0 would be by far the most common velocity.

- Open the table editing windows containing the histograms of pitches and velocities, and play on the synth to see how the distributions are stored.

Rhythm Analysis

In the patch we use a simple method of rhythmic analysis to keep a histogram of the rhythms played. We use a **timer** to get the time between note-ons, and divide the time by 30 to get the *rhythm*—the number of 30ms pulses that elapse between notes.

If the time between notes is less than 1 pulse (30ms), we assume the second note is virtually simultaneous with the previous note and should therefore not be included in our analysis. If the time between notes is greater than 96 pulses (2880ms), we assume that the performer has stopped playing momentarily, or is holding an extremely long note. In either case, we don't want to include it in our histogram. So the **split** object passes only rhythms that are between 1 and 96 pulses in length, and a histogram of these rhythms is stored in the **table**.

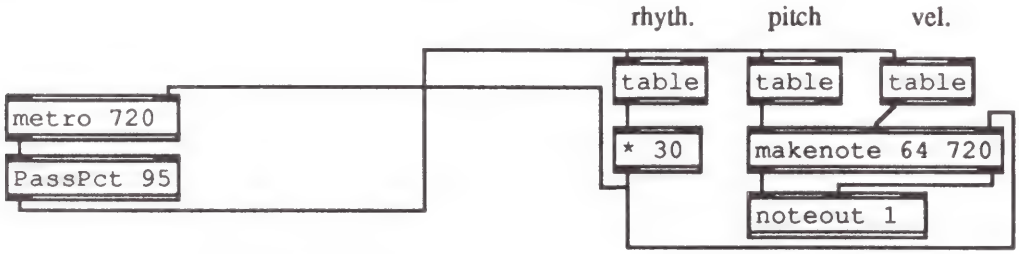


Tutorial 33

Probability tables

An Improviser with a Memory

When the **metro** in the bottom left corner of the Patcher window gets turned on, it sends bangs to **PassPct** (the patch we wrote in Tutorial 29), and 95% of the bangs get passed on to the three **tables**. A velocity, a pitch, and a rhythm are sent out, with the choice of each being based on the stored probability distributions (the histograms of what has been played by the performer). The velocity and the pitch are sent immediately to the synth. The rhythm is translated back into milliseconds by multiplying it by 30, then it is sent to the **metro** to set a new speed (and to **makenote** to set a new duration for the subsequent note).



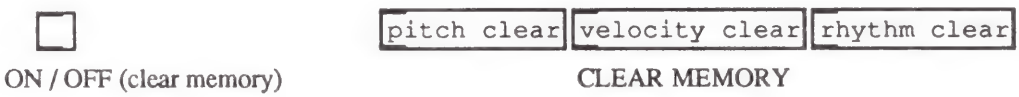
The resulting "improvisation" bears some resemblance to what you played on your MIDI keyboard, because it uses the same pitches, velocities, and rhythms, but the improviser patch recombines these parameters randomly. Because of the **PassPct** object, the improviser also rests about 5% of the time.

The User Interface

We had to decide how much control the performer should have over the improvising patch, and how the control should be implemented. We decided that the improviser would be turned on by moving the modulation wheel to any position other than 0, or by clicking on a **toggle** object.

We also wanted the performer to be able to erase the improviser's memory, either all parameters or just one parameter, so that its memory can be filled with new information. This requires sending clear messages to the **Histo** and **table** objects, to set all their values to 0.

We decided to have all mouse controls located in a separate window, and have automatic on/off control from the mod wheel as well. We have hidden most of the objects and patch cords in the *[controls]* subpatch window, so if you want to see how the main patch communicates with the subpatch you'll need to unlock the *[controls]* window.



ON / OFF (clear memory)

CLEAR MEMORY

The data from **ctlin** is sent to the **toggle** in the subpatch, then back to the main patch. This lets us use the **toggle** both for displaying the on/off state received from the mod wheel and for actually *sending* on/off commands with the mouse.

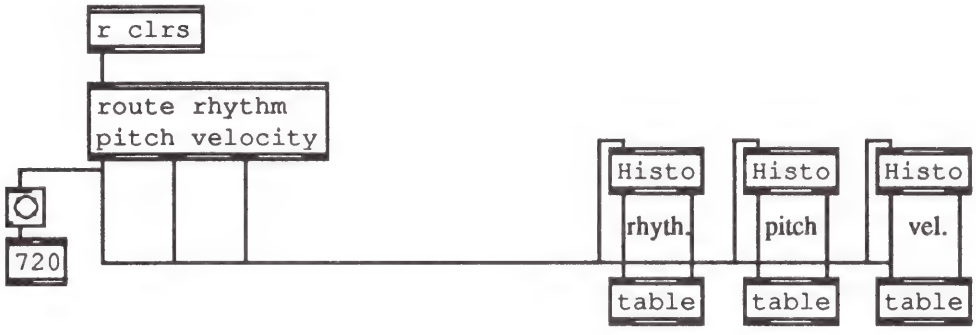
Tutorial 33

ON / OFF (clear memory)

The on/off state (0 or non-zero) is sent to **TogEdge**. **TogEdge** sends a bang out one of its outlets only when the number it receives represents a change from 0 to non-zero or vice versa. The left outlet is for changes from 0 to non-zero and the right outlet is for changes from non-zero to 0. If we sent the control data directly to the **metro**, the **metro** would get restarted with *every* non-zero number from the mod wheel. **TogEdge** lets us detect only the essential control data: changes to and from 0.

Note: A few objects such as **Histo** and **TogEdge** contain upper case letters in their names. When typing one of these names in an object box, be sure you enter the name correctly because Max *does* distinguish between upper and lower case letters.

When **TogEdge** receives the *on* status from the **toggle**, it turns on the **metro**. When it receives a 0, it turns off the **metro** and sends a bang to all the message boxes in the controls window. Each of the clear messages is routed to the proper **Histo** and **table** objects with **route**. Clearing the rhythm also resets the time of **metro** and **makenote** to 720.



The reason we used three different message boxes to send the clear messages separately is because it also gives the user the option of clearing the memory of only *one* parameter by clicking on a specific message box. Turning off the improviser clears *all* memories at once.

If we really wanted to make this improviser patch into a completed Max program for someone ese to use, we would probably hide everything except the controls (plus a few comments to tell the user what to do). We left most things visible here so you could examine the patch.

Summary

Histo keeps an internal *histogram* of the numbers it has received. When it receives a number in its left inlet it adds the number to its internal histogram, sends a report of how many times it has received that number out the right outlet, and sends the number itself out the left outlet.

The output of **Histo** can be sent directly to a **table**, so that the frequency of occurrence of each number, as reported by **Histo**, is stored as a value in the **table**. You can open the graphic window of the **table** to see the histogram.

Tutorial 33

Probability tables

A clear message in the left inlet of **Histo** or **table** sets all values to 0. A bang in the left inlet of **table** causes it to send out an *address* rather than a value. The probablility of a specific address being sent out depends on the value it stores, compared to the other values in the **table**. The greater the stored value of an address, the more likely that address is to be sent out when a bang is received. This feature of **table** allows you to use it for probability distributions.

By sending bang to a **table** that contains a histogram (a frequency distribution of past numbers, received from **Histo**), you can cause numbers to be sent out of the **table**, with the likelihood of a getting a number based on how frequently it has occurred in the past.

TogEdge is used to detect a *change* in the zero/non-zero status of incoming numbers. When the numbers change from 0 to non-zero, a bang is sent out its left outlet; when the numbers change from non-zero to 0, a bang is sent out its right outlet.

Using **route** to detect specific *selectors* (the first item in a message), messages can be routed to different destinations.

See Also

Histo	Make a histogram of the numbers received
Quantile	Using **table** for probability distribution
table	Store and graphically edit an array of numbers
Tables	Using the **table** graphic editing window
TogEdge	Report a change in zero/non-zero values

Managing raw MIDI data

Tutorial 34

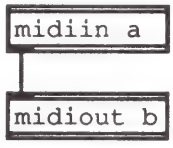
midiin and midiout

MIDI objects such as **notein**, **noteout**, **bendin** and **bendout**, send and receive specific types of MIDI data. If you want to send or receive *all* types of MIDI data as individual bytes (including status bytes), use **midiin** and **midiout**.

The **midiin** object is useful for examining every incoming MIDI byte. As we will see in Tutorial 35, it is also used for recording MIDI from your gear into the sequencer object, **seq**. The **midiout** object is used for sending any type of MIDI message to the synth, including system exclusive messages. It is also used to send MIDI data that is played back from the **seq** object.

In the simplest possible situation, Max can turn your computer into a very expensive MIDI thru box, by simply connecting the outlet of **midiin** to the inlet of **midiout**. These two objects can be given a letter argument specifying a port through which to receive or transmit, so you can use them to route MIDI data from one port to another. If there is no argument, port a is assumed by default.

What comes in port a is echoed back out port a.

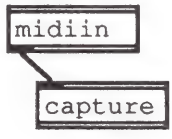


What comes in port a is transmitted out port b.

For more information about port assignment, look under Ports in the *Topics* section.

capture

If you just want to examine the MIDI bytes that your equipment is sending out, you can connect the outlet of **midiin** to a **capture** object, as we have done in this patch.



The **capture** object is a good all-purpose debugging tool. It collects the numbers it receives, and when you double-click on it, opens a Text window for you to view the numbers. The numbers stored in **capture** are not saved when the patch is closed, but you can save the Text window as a separate file or copy the numbers and paste them in somewhere else—into another text window, or even into a graphic table window. Whenever you want to see what numbers are being sent from an outlet, just connect the outlet to a **capture** object, run the patch, then view the contents of **capture**.

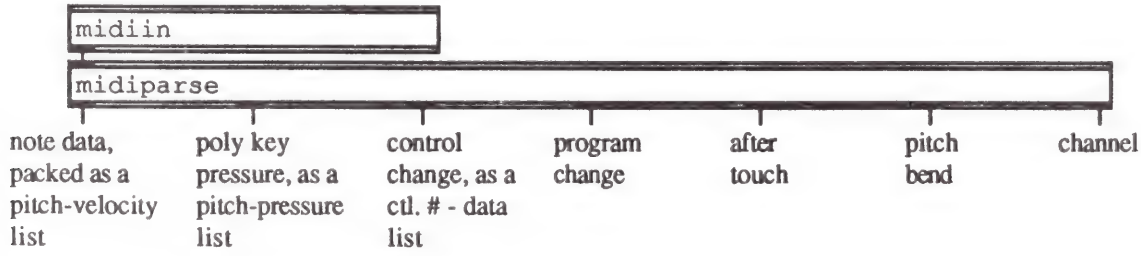
- Send out various types of MIDI messages from your keyboard: pitch bend, modulation, notes, program changes, etc. Every byte is received by **midiin** and stored in **capture**. Double click on the **capture** object to see the MIDI data.

Tutorial 34

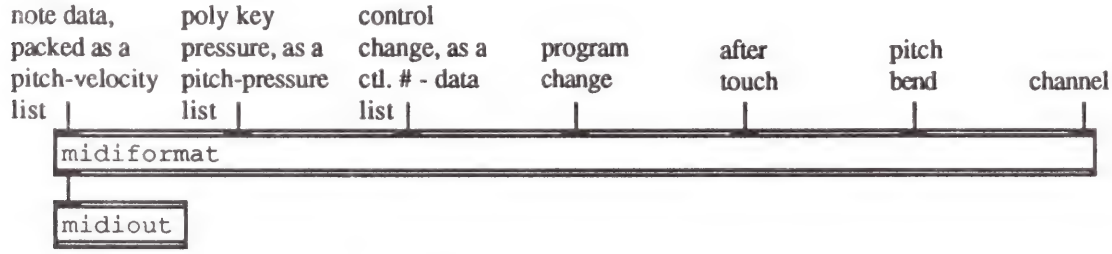
Managing raw MIDI data

midiparse and midiformat

The **midiparse** object sorts the raw MIDI data it receives from **midiin**, and sends the sorted data out its outlets. The combination of **midiin** and **midiparse** is like having all of the specialized MIDI receiving objects in one place.



The **midiformat** object performs exactly the reverse function of **midiparse**. It prepares data into well-formatted MIDI messages with the appropriate status byte, and sends each byte to **midiout** for transmission to the synth.

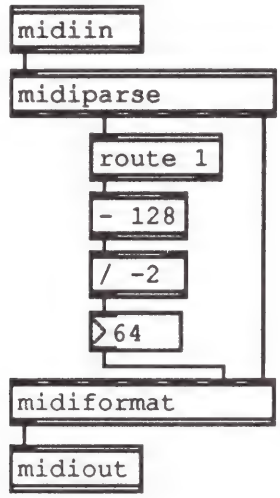


Parsing and Formatting MIDI Data

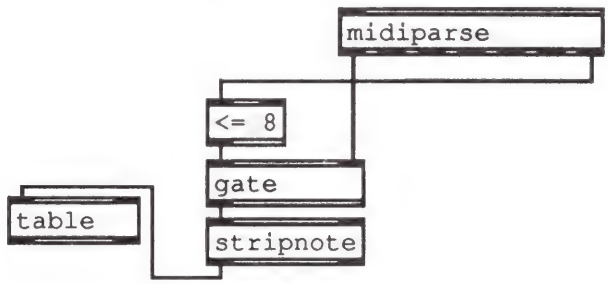
In the example patch, we have shown a couple of ways in which diverse MIDI data from **midiparse** may be used to control objects in Max, or may be given another meaning and transmitted with **midiformat** and **midiout**.

The control data from the third outlet of **midiparse** is sent to **route**, which selects only data from controller 1, the mod wheel. The mod wheel data, from 0 to 127, is mapped to the range 64 to 0, then it is reassigned as pitch bend data by **midiformat** and transmitted to the synth. The resulting effect is that the mod wheel of the keyboard also controls the pitch bend. As the modulation increases from 0 to 127, the pitch is bent downward from 64 to 0. This type of reassignment is a convenient way of correlating two different kinds of control data.

Tutorial 34



Another part of the patch shows how you can select data from a *group* of MIDI channels. The channel number is used to open or shut a **gate** for the note data. Only note data on channels 1 through 8 is sent on, and the pitch data triggers a number from the **table**.



- Play notes on your MIDI keyboard and you will hear that each note-on pitch is also used as an address to trigger a value from the **table**. If you set your keyboard to transmit on a channel between 9 and 16, the notes will not be passed by the **gate**.

Copying Captured Values into a table

The incoming pitch bend data is sent out of **midiparse** to a **capture** 128 object. The argument to **capture** sets the quantity of numbers it will store. This is one way to produce values for a **table** quickly and easily. It's a good way to preserve something you have done, such as a nice pitch bend, and save it in a **table** for future use.

To copy captured pitch bend values into a **table**:

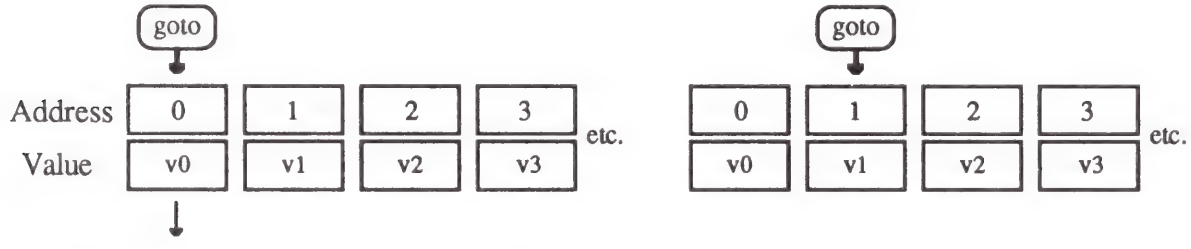
1. Click on the clear message to clear the **capture** objects.
2. Move the pitch bend wheel for at least 3.2 seconds. (The **speedlim** object limits the incoming pitch bend values to 40 per second.) After 3.2 seconds, the earliest values received by **capture** will be lost as new ones are received.

3. Double-click on the **capture** 128 object to open its Text window.
4. Choose **Select All** from the Edit menu.
5. Choose **Copy** from the Edit menu.
6. Close the Capture window.
7. Double-click on the **table** object to open its graphic editing window.
8. Choose the *Selection tool* from the table window palette.
9. Choose **Select All** from the Edit menu.
10. Choose **Paste** from the Edit menu.
11. If you want to, you can save the Table window as a separate file, for future use in patches.

Stepping Through a table

In the left part of the Patcher window we introduce another way to step through the values in a **table**. A **table** object has a pointer—a place in memory where it stores an address. You can set the pointer to point at any address in the table with the word goto, followed by the address number in the left inlet. For example, the message goto 0 sets the pointer at address 0 in the table, the first address.

When the message next is received in the left inlet, **table** sends out the value stored at the address at the pointer, then increments the pointer to the next address. When the pointer reaches the last address in the **table**, a next message will cause it to *wrap around* and point to the first address again, so you can use next to cycle continuously through a **table**. (You can also cycle backward through a **table** with the prev message, not shown here.)

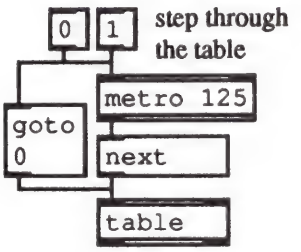


A next *message sends out the value in the address at the pointer...*

...then sets the pointer to the next address

Thus, in our patch values are sent out of the **table** each time a pitch is played on your keyboard (on channels 1 through 8), and values can also be sent out automatically by turning on the **metro** to send repeated next messages to the **table**.

Tutorial 34



System Exclusive Messages

MIDI *system exclusive* (*sysex*) messages are used to send information other than that which is established as standard by the MIDI specification. Sysex commands are implemented by manufacturers as a way of modifying settings on their gear via MIDI.

Max has a **sysexin** object for receiving system exclusive messages, but to *send* sysex messages you need to format them yourself and then send them using **midiout**.

There is an object to help you format sysex messages, called **sxformat**. The **sxformat** object lets you specify some bytes of a MIDI message as arguments, and other bytes as *changeable arguments* to be replaced by numbers received in the inlet(s).

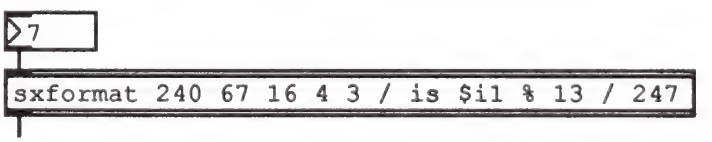
The format of changeable arguments in an **sxformat** object is different from that of changeable arguments in a **message** object. Changeable arguments in **sxformat**:

- contain the letter i to indicate that they are integer arguments (as in $i1)
- are preceded by the word is
- are bounded on either side by a slash (/)

For example, the changeable argument / is $i2 / will be replaced by the number received in the second inlet (or the second number in a list received in the left inlet).

Calculations can even be performed on incoming numbers using the changeable argument. For example, the changeable argument / is $i1 + 1 / adds 1 to the number received in the left inlet before sending it out.

When **sxformat** receives a number or a list in its left inlet, it uses the number(s) to replace any changeable arguments, then sends each of the arguments out the outlet in sequential order.

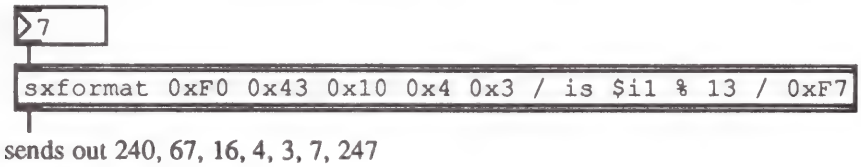


sends out 240, 67, 16, 4, 3, 7, 247

Tutorial 34

Managing raw MIDI data

Programmers often express bytes of a sysex message in *hexadecimal* format, rather than decimal. If you prefer to type hexadecimal numbers, you can do so in Max by preceding the hexadecimal number with 0x (zero-x). Here is an example of the same sysex message expressed in hexadecimal:

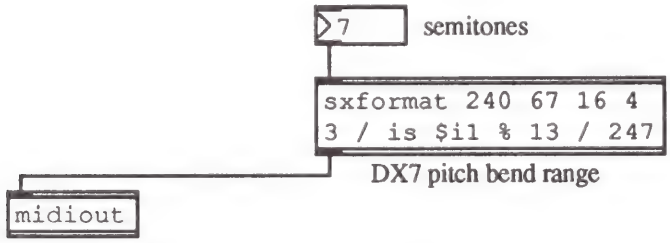


An Example Sysex Message

The status (beginning) byte of any sysex message is always 240. The second byte is the Manufacturer ID; each major synthesizer manufacturer has a unique number asssigned to the brand name. The next bytes are established by the manufacturer—as many as are needed to express whatever is being expressed. A sysex message always ends with the "end-of-sysex-message" byte, 247.

When a synthesizer receives the sysex status byte, 240, it looks at the second byte. If the second byte is the ID of some *other* manufacturer, the synth ignores all the subsequent bytes until it sees 247. Then it begins to pay attention to incoming MIDI messages again.

In the bottom-right corner of the Patcher window is an example of the use of **sxformat**. It is designed to change the effective pitch bend range on a Yamaha DX7 synthesizer (or TX sound module). The first argument is the sysex status byte, 240, and the second argument is the Manufacturer ID for Yamaha, 67. Yamaha decided that the next byte would tell the synth what kind of message it's going to receive; in this case, 16 means "parameter change on channel 1." The fourth byte specifies that the sysex message is a *performance parameter* change. The next byte is the parameter number—3 is for *pitch bend range*.



The next byte specifies the setting for the pitch bend range—how many semitones up and down we can bend the pitch. This is the value we want to be able to change, so we've made this byte a changeable argument in **sxformat**. The pitch bend range value must be from 0 to 12 semitones, so we've included a % 13 calculation to limit incoming numbers between 0 and 12: / is $i1 % 13 /. That's the end of the data portion of the message, so the ending byte, 247, comes next.

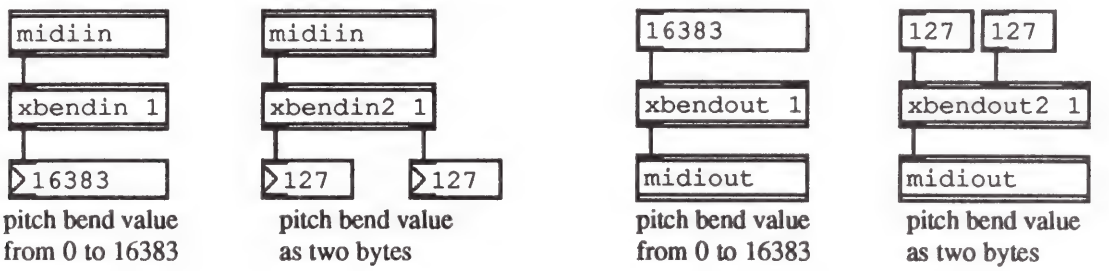
When a number is received in the inlet, the entire message is sent out, one number at a time, using the incoming number as the *pitch bend range* value.

- If you have a Yamaha DX7, you can change the pitch bend range by dragging on the **Number box**. If your don't, your synth will ignore this message.

Extra Precision Pitch Bend Data

Most MIDI keyboards transmit and receive 128 different pitch bend values, and Max's MIDI objects do the same. However, a MIDI pitch bend message actually contains another byte for additional precision in expressing the pitch bend amount, and some synthesizers take advantage of this capability. If a synth does *not* have the extra precision capability, it always transmits a value of 0 in the extra precision byte, and ignores the extra byte when it is receiving pitch bend messages.

For MIDI keyboards that do have the extra precision capability, Max has objects for intepreting incoming extra precision pitch bend data received from **midiin**, and for formatting extra precision pitch bend messages to be transmitted by **midiout**.

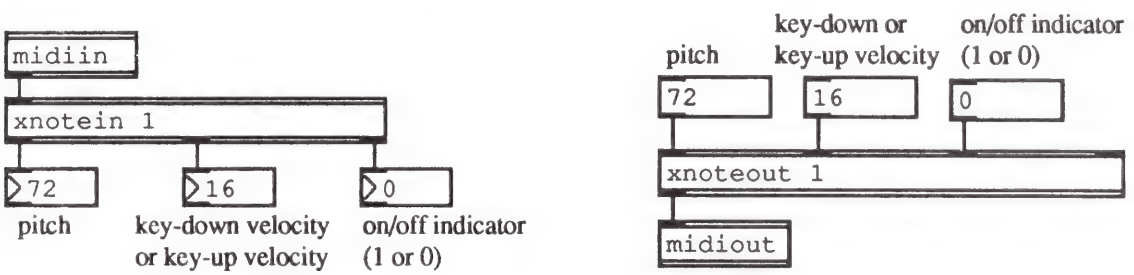


Because relatively few MIDI instruments have this capability, we won't discuss the matter in detail in this Tutorial. For more information, look under **xbendin** and **xbendout** in the *Objects* section.

Note-Off Messages with Release Velocity

In MIDI there are two ways to express a note-off. One way is as a *note-off* message with a release (key-up) velocity, and the other way is as a *note-on* message with a key-down velocity of 0. Since most synths are not sensitive to key-up velocities, **noteout** uses the latter method for note-offs.

For synths that are sensitive to key-up velocity, however, Max has objects for interpreting and formatting note-off messages with release velocity. To read more about these objects, look under **xnotein** and **xnoteout** in the *Objects* section.



Tutorial 34

Managing raw MIDI data

Summary

The **midiin** object ouputs each byte of MIDI data it receives. The **midiout** object sends any number it receives in its inlet. You can set these objects to send or receive through a specific port by typing in a letter argument to specify the port.

The **midiparse** object interprets raw MIDI data from **midiin** and sends each type of data out a different outlet. The counterpart to **midiparse** is **midiformat**, which receives data in its various inlets and prepares different types of complete MIDI messages, which are sent by **midiout**.

To aid you in formatting system exclusive messages to be sent by **midiout**, **sxformat** lets you type in arguments which it sends out one at a time as individual bytes. You can include changeable arguments in **sxformat** which will be replaced by incoming numbers before the message is sent out.

The **capture** object stores a list of all the numbers it receives. You can view the list in a Text window by double-clicking on the **capture** object, and you can copy the contents of that Text window into a Table window. The **capture** object is good for viewing any stream of numbers when you are trying to figure out exactly what numbers are coming out of an outlet.

See Also

Librarians	Creating patch editors and librarians
MIDI	MIDI software protocol
midiformat	Prepare data in the form of a MIDI message
midiin	Ouput incoming raw MIDI data
midiout	Transmit raw MIDI data
midiparse	Interpret raw MIDI data
Ports	How MIDI ports are specified
sxformat	Prepare MIDI system exclusive messages
xbendin	Interpret extra-precision MIDI pitch bend messages
xbendout	Format extra precision MIDI pitch bend messages
xnotein	Interpret MIDI note messages with release velocity
xnoteout	Format MIDI note messages with release velocity

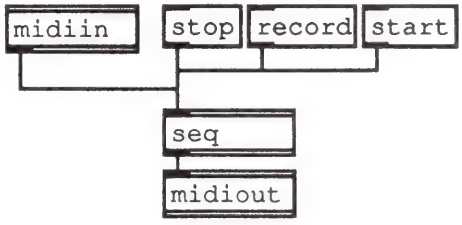
seq *and*
follow

Tutorial 35

seq

Max has three objects for recording and playing back MIDI performances: **seq**, **follow**, and **mtr**. In this chapter of the Tutorial we will discuss how to record a single track of MIDI data with the basic sequencing object **seq**, and how to compare a live performance to a previously recorded performance—in order to follow along with a performer—using **follow**.

The **seq** object records and plays back raw MIDI data in conjunction with **midiin** and **midiout**. It understands various text messages to control its operation, such as stop, start, and record.



Patch 1 contains the basic **seq** configuration shown above, plus a few other useful messages.

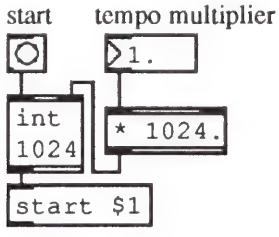
- Click on the record message box in Patch 1. Play notes, pitch bends, and modulation on your MIDI keyboard. Click on the word start to hear your performance played back. (You don't need to click on stop first, because start automatically stops the recorder before playing back.)

There is probably a delay before you what you played, because you didn't start playing at exactly the same moment you sent the word record to **seq**. The delay message can be used to change the starting time of the sequence.

- Click on the message containing delay 0 to set the starting time of the sequence to 0. Now when you click on start again, the sequence starts playing immediately.

The start message can be followed by a number argument specifying the tempo at which you want the sequence to be played back. The start argument divided by 1024 determines the factor by which the tempo will be increased or decreased. So, for example, the message start 1024 indicates the original (recorded) tempo, the message start 1536 plays the sequence back 1.5 times as fast, start 512 plays it back half as fast, and so on.

In the upper-left corner of Patch 1 we've devised a way to calculate the tempo ratio, letting you specify the tempo in terms of a multiplier, with 1 being the normal tempo.



- Drag on the **Number box** to choose a tempo multiplier. (You can change the fractional part of the number by dragging with the mouse positioned to the right of the decimal point.) Then click on the **button** to play your sequence at the new tempo.

Changing the playback speed in this manner does not actually change the times recorded in the sequence, it merely changes the speed at which **seq** reads through it. Another message, called hook (not shown here), alters the times in a sequence. Look under **seq** in the *Objects* section for details.

Saving and Recalling Files

If you like, you can save the sequence you have recorded in a separate file, to be used later. The write message opens the standard **Save As...** dialog for you to name the file where you want to store the sequence. You may want to give the names of your sequence files some unique characteristic so you can distinguish them from Patcher files and Table files. (We use *.sc* at the end of the name to identify the file as a musical score).

If you check *Save as Text* in the dialog box when you save the file, you can view the sequence in a Text window by choosing **Open As Text...** from the File menu. Otherwise, the file is stored as a standard MIDI file.

To load a saved file into a **seq** object, send the read message to **seq**, and a standard **Open...** dialog will appear so you can choose the file you want to load in. If the read message is followed by a file name argument, **seq** loads that file automatically (provided it's located where Max can find it). You can set **seq** to load a file automatically when the patch is opened by typing the name of the sequence file as an argument to the **seq** object.

- Click on the **message** containing read bourrée.sc to load in a brief melodic excerpt from a Bach bourrée in E-minor. Send a start message to **seq** to hear the melody.

Processing a MIDI Sequence

The output of **seq** is in the form of individual bytes of MIDI messages, and can be transmitted directly to the synth with **midiout**. It can also be sent to **midiparse**, however, and the parsed data can then be processed by other Max objects before it is sent to the synth.

In the **patcher** transpose object we parse the raw MIDI data received from **seq**, transpose the pitch of the notes by some amount, then reformat the MIDI messages and send them to **midiout**.

- Double-click on the **patcher** transpose object to see its contents.

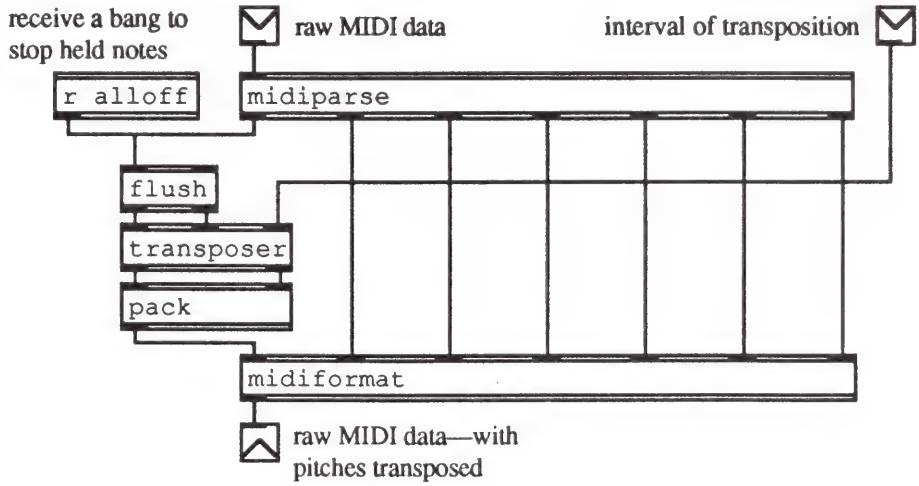
The **patcher** transpose subpatch contains a *nested* subpatch, the **transposer** patch that we made in Tutorial 27. Subpatches can be nested in this manner so that each task of a patch is *encapsulated* and is easily modified. (For more on this subject, look under *Encapsulation* in the *Topics* section.)

Notice that we have included an additional handy feature inside the **patcher** transpose subpatch: a **flush** object to turn off held notes. When **seq** is playing a sequence and gets stopped by a stop message, it may be in the middle of playing a note, and the note-off message will not be sent out.

seq *and* **follow**

Tutorial 35

In Patch 1, we made the stop message also trigger a **button** which sends a bang to the **flush** object in the subpatch to turn off any such stuck notes.



In general, whenever your patch is capable of stopping **seq** while notes are being recorded or played back, there is the potential for vital note-off messages to be lost. This is especially true if your patch sends stop, record, or play messages by some automatically generated means. Bear this potential danger in mind when constructing your patch, and include an object such as **flush**, **poly**, or **makenote**—whichever is appropriate—to provide missing note-offs. Examples are shown in Tutorial 13.

- Record a sequence (or use the bourrée excerpt), and play the sequence with a start message. Try changing the transposition with the **hslider** while the sequence is playing.

follow

The **follow** object is very similar to **seq** in its ability to record MIDI data. But whereas **seq** only records MIDI messages, **follow** can also record a sequence of single numbers which are not in the form of complete MIDI messages (such as the pitches from MIDI note-ons).

follow *can record MIDI messages, or single numbers (e.g., just note-on pitches)*

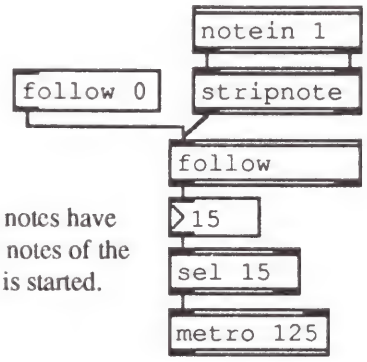
A sequence can be stored in **follow** by recording MIDI data, by recording a series of single numbers, by reading in a file with a read message, or by typing in a file name argument. Once it

Tutorial 35

seq *and* **follow**

has a stored sequence, **follow** can use that sequence as a musical score, and follow along while a performer plays the music. Each time the performer plays a pitch that matches the next note-on in the stored sequence, **follow** sends the pitch out its right outlet and sends the index number of that note's position in the sequence (1, 2, 3, etc.) out its left outlet.

The particular utility of this score-following feature is that the index numbers can be used to trigger other notes, or any other process such as, say, turning on a **metro** when the 15th note is matched.



When the incoming notes have matched the first 15 notes of the sequence, the metro is started.

How follow Follows

When **follow** receives the message follow with a number argument, it begins to look for incoming pitches which match the notes in the score, starting at the index specified in the argument. For example, follow 10 causes the object to look for incoming pitches that match the 10th note in the score. When the matching pitch is received, **follow** sends that pitch out its right outlet, and sends the index out its left outlet.

The **follow** object even allows for wrong notes, so if the performer plays a couple of spurious notes, or skips a note or two, **follow** will still be able to keep track of the performer's progress through the score.

One can also step through the score with repeated next messages. After a follow message has been received, the message next triggers the pitch at the specified index and increments the pointer to the next index.

An Attentive Accompanist

When we use the index numbers from the left outlet of **follow** as addresses of a **table**, or addresses of some other array object like **funbuff**, the index numbers can trigger other values. In this way, we can create an accompanist who "knows the score" and follows along with the performer. Each time the performer plays a note of the score, the accompanist has a specific reaction—play a simultaneous note or notes, play some independent melody, rest, whatever—and seems to follow along with the performer.

We've made such an accompanist in Patch 2. The accompanist plays the left hand part of the Bach E-minor bourrée while you play the right hand part. The **follow** object has loaded the sequence file

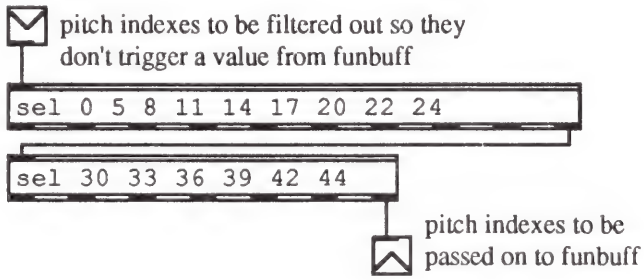
bourrée.sc to use as the score. Each time a note of the score is played, an index number is sent out that triggers some sort of reaction.

- Click on the follow 0 message to start the score-follower at the beginning of the score. Play the right hand part of the bourrée excerpt and Patch 2 will play the left hand part along with you.
- If you've forgotten how the melody goes, read the *bourée.sc* sequence into the **seq** object in Patch 1 and listen to it.
- Click on follow 0 again, and play the melody with an occasional wrong note or skipped note. If you don't mess up too much, **follow** manages to account for your mistakes and continues following the score.
- Try the melody again, with ritards at the end of the phrases. The extra notes that the accompanist plays match your tempo.

Analysis of Patch 2

Sometimes we want the left hand to play a note along with the right hand, other times we want the left hand to do nothing new (when the right hand is playing the second of a pair of eighth notes and the left hand is just holding a quarter note), and occasionally we want the left hand to play a note *in between* notes played by the right hand. How do we accomplish each reaction?

The index numbers are first sent to a subpatch called **patcher** silencer. This subpatch simply filters out the index numbers which we don't want to trigger a note of the accompaniment. The **sel** objects select those index numbers and pass the rest on.


pitch indexes to be filtered out so they don't trigger a value from funbuff
sel 0 5 8 11 14 17 20 22 24
sel 30 33 36 39 42 44
pitch indexes to be passed on to funbuff

Contents of the **patcher** silencer *subpatch*

Notice that **sel** objects can be linked together to select more than 10 numbers, since the numbers which are not matched by the first **sel** object are passed out the rightmost outlet to the second **sel** object.

The remaining index numbers are sent as addresses to **funbuff**, which sends out an appropriate accompanying pitch value. To make **funbuff** respond properly, we simply made a list of *addresses* and *values* and saved the list as a **funbuff** file named b*ourrée.fb*.

- If you want to see the contents of **funbuff**, choose **Open As Text...** from the File menu and open the file named *bourrée.fb*.

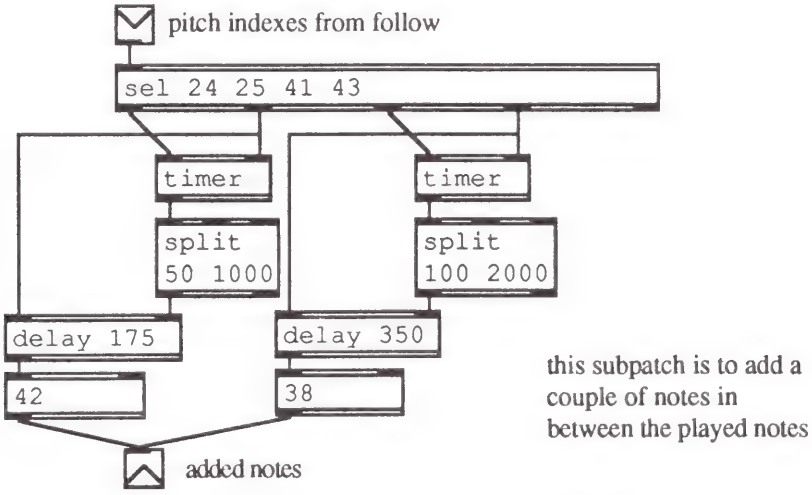
Tutorial 35

seq *and* **follow**

We could have also stored the accompaniment pitches in a **table**—or in a **coll** object, which will be explained in Tutorial 37.

So far we have made the accompanist play some notes that are simultaneous with the melody notes, and we've made the accompanist rest on melody notes which are unaccompanied, but how about when the accompanist has to play notes on its own, in between melody notes? This occurs twice in the score, once at the end of each phrase.

To help the accompanist play notes on its own, the **patcher** addnotes object measures the tempo of the performance and plays notes with a delay time based on its perception of the performer's tempo.



Contents of the **patcher** addnotes *subpatch*

For example, the subpatch measures the amount of time between notes 24 and 25 of the melody (the speed of an eighth note), then delays for that amount of time before triggering the pitch 42. Likewise, the time between the 41st and 43rd melody notes (the speed of a quarter note) is used to as a delay time before sending out the pitch 38. This is a simple (but fairly effective) method of analyzing the performer's tempo and playing notes in that tempo.

It's always a good idea in programming (and elsewhere, for that matter) to prepare for the unexpected. What happens if the performer accidentally misses one of these notes that we need to analyze the tempo and trigger added accompaniment notes?

If the performer misses the first note of a pair, for example, the second note will trigger a ridiculously large value from the **timer** and the accompaniment note will get delayed far too long. To protect against this eventuality, we have used **split** objects to limit the time values that can be sent to **delay** within certain (only *moderately* ridiculous) extremes. If the value from **timer** exceeds these limits, the **delay** object will use the delay time in its argument. If the performer misses the second note of a pair but continues on, the added note will never get played, but by then the performer will have passed that point anyway, and **follow** will keep up with the performer.

seq *and*
follow

Tutorial 35

The pitches from **patcher** addnotes and from **funbuff** are sent to **makenote** where they are paired with the velocity of the right hand melody notes, so the accompanist is sensitive to the performer's dynamics, as well. Rather than use an algorithm or a lookup table to provide durations for the acompaniment notes, we just picked a duration that seems to work both as an eighth note duration and as a stylistically staccato quarter note.

Summary

A single track of raw MIDI data can be recorded and played back (at any speed) with the **seq** object. The MIDI data is received from **midiin** and is sent by **midiout**. You can also parse the data from **seq** using **midiparse**, and process the numbers with other Max objects before transmitting them.

A recorded sequence can be saved as a separate file by sending a write message to **seq**. If you check the *Save as Text* option in the **Save As...** dialog box, you can open and edit the file later with **Open As Text...**. A MIDI file can be read into **seq** by sending a read message, or by typing in the file name as an argument.

The **follow** object allows you to record or read in a sequence, then use that sequence as a musical score to follow along with a live performance. As the pitches received in the inlet are matched with notes in the score, the index number for each note is sent out, and can be used to trigger other notes or processes.

See Also

follow	Compare a live performance to a recorded performance
mtr	Multi-track sequencer
seq	Sequencer for recording and playing MIDI
Sequencing	Recording and playing back MIDI performances

Tutorial 36

Multi-track sequencing

mtr

The **mtr** object is Max's most versatile sequencer. It can record and play back up to 32 different tracks of messages: numbers, lists, or symbols. The tracks can be recorded and played back either separately or all together. With this versatility, you can record and play back not only MIDI bytes, but numbers from any object such as a **slider** or a **dial**, sequences of text messages to be displayed to the user, pitch-velocity lists, etc.

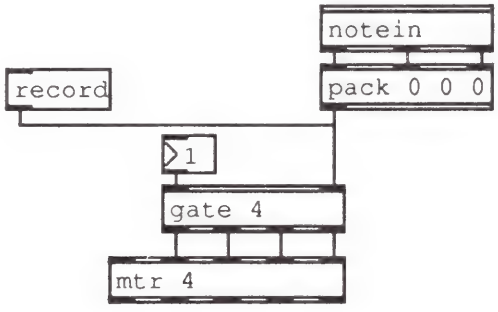
We'll show how **mtr** is used to record and play back MIDI data.

The number of tracks in an **mtr** object is specified by a typed-in argument. The leftmost inlet is a *control inlet* for receiving commands, and the other inlets are for messages you want to record. The command messages for **mtr** are similar to those for **seq**, but not identical. Notably, **mtr** undertands the message play as well as start, and the play message does not take a tempo argument.

When command messages such as stop, play, record, mute, and unmute are received in the left inlet they apply to *all* tracks of **mtr**. These commands can be followed by a number argument, however, specifying a unique track to which the message applies. Alternatively, these messages can be received in an individual track's inlet, to give a command to just that track.

A 4-Track "Simul-Sync" Recorder

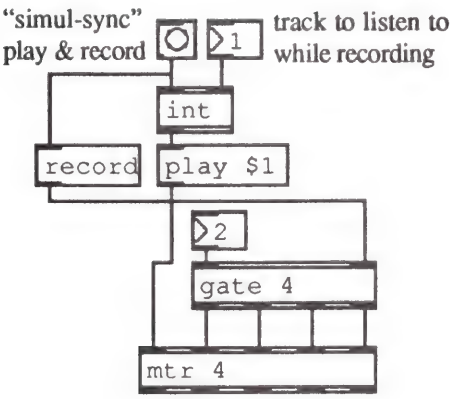
Patch 1 shows a configuration to record four separate tracks of MIDI note data separately, then play them all back together. **Number boxes** let you specify the track you want to record on and, if you wish, a track to listen to while you are recording. When you choose a track to record, the **gate** opens that outlet to let the record message and the note data go only to that track.



- Set the open **gate** outlet to 1 (to record on track 1), and click on the record message. Play some notes on your MIDI keyboard. When you are finished recording, click the play message to hear what you have recorded.

- Now open gate outlet 2, and enter the number 1 in the **Number box** at the top of the patch so that you can listen to track 1 while you record track 2.

Tutorial 36

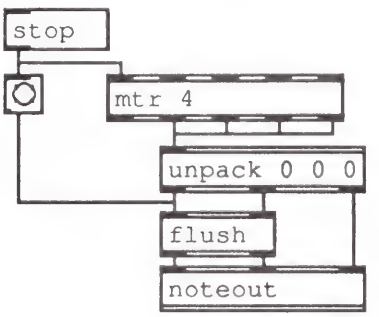


When you click the "simul-sync" **button**, the message play 1 will be sent to the left inlet of **mtr**, and the message record will be sent to the inlet of track 2.

- Click the **button** and record track 2. When you have finished, click on play to hear both tracks.

You can continue in this manner to record all four tracks. If there is some delay between the time you click play and the time the sequence starts to play, it's because you took some time to begin recording notes after you clicked record. To eliminate this delay, and cause the first event in **mtr** to begin at time 0, click the message first 0.

Notice that once again we have included a **flush** object to guard against stuck notes. Every time a stop message is sent to **mtr**, a bang is also sent to **flush** to turn off any notes currently being held.



Recording Messages from Different Sources

Note data is not the only thing that can be recorded with **mtr**; messages from virtually any combination of objects can be recorded and played back by the same **mtr** object. In Patch 2 we record numbers from **pgmin**, **bendin**, **ctlin**, and a **dial**, each on a different track.

Tutorial 36

Multi-track sequencing

- Click on the record message in Patch 2 and send pitch bend, modulation, and program change messages from your MIDI keyboard for several seconds. You can also move the **dial** with the mouse. When you have finished, click play and you will *see* your performance played back, controlling other objects.

The first 0 message can be used to eliminate the delay between the time you clicked record and the time you started to transmit MIDI messages from the keyboard. The message delay 0 causes *every* track to start at time 0, even if you *started* sending data to the tracks at different times.

- To see the difference between first 0 and delay 0, click on record and send about 5 seconds of pitch bend data, then 5 seconds of mod wheel data, and so on. When you have finished, click stop.

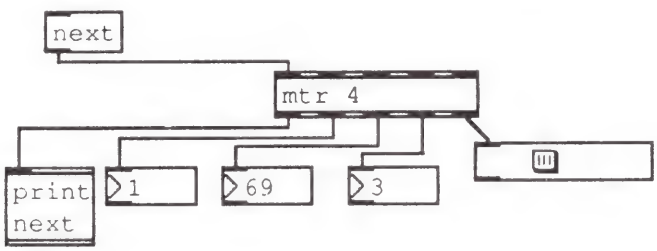
- Next, click first 0 to eliminate the initial delay before any data was recorded. Click play to see your performance replayed.

- Now click delay 0 and play your sequence again. This time all tracks start at time 0, even though you started recording data on one track before the others.

When you send **mtr** a mute message while it is playing, it continues to play its stored sequence, but it *suppresses* the actual output. Use the unmute message to restore output. Individual tracks can be muted and unmuted by following the mute or unmute message with a track number argument, or by sending the messages into a specific track's inlet.

You can *step through* the messages stored in **mtr** by sending repeated next messages to the control inlet. When **mtr** receives next, it sends out the next message stored in each track. It also sends a two-item list out the leftmost outlet once for each track, reporting the track number and the duration (the time between that message and the following one in the track).

- Check the **All Windows Active** command in the Options menu, and bring the Max window to the front so you can see what gets printed in it. Then click on the next message. The next value stored in each track is sent out the track outlets, and a list for each track, consisting of the track number and the duration between messages, is sent out the left outlet.



The rewind message is used in conjunction with next. It sets the pointer back to the beginning of the sequence, so that the message next will start at the beginning again.

Multi-track sequencing

Tutorial 36

Summary

The **mtr** object records and plays back up to 32 tracks of any message type—numbers, lists, or symbols. Tracks can be recorded, played, stopped, muted, and unmuted—either individually or all tracks at the same time.

The next message can be used to *step through* the recorded messages instead of playing them back at their original recorded speed.

See Also

follow	Compare a live performance to a recorded performance
mtr	Multi-track sequencer
seq	Sequencer for recording and playing MIDI
Sequencing	Recording and playing back MIDI performances

Tutorial 37

Data structures

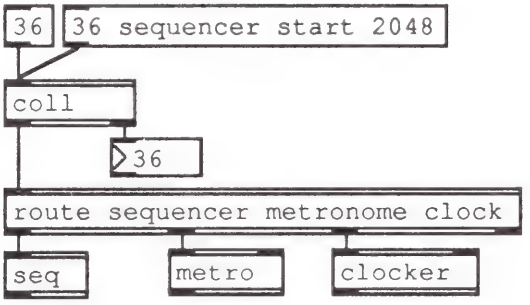
What Is a Data Structure?

A *data structure* is any collection of data that is stored in some arrangement that allows individual items to be found easily. In Tutorial 32, we used the **table** object, a data structure called an *array*, where we used an index address to access the stored values. In this chapter, we'll use some objects that allow you to create your own collections of data and retreive them with whatever addresses you wish.

coll

The most versatile data structure object in Max is the **coll** (short for *collection*). A **coll** object stores a collection of many different messages, of any type and length (up to 256 items long), and can give each message either a number address or a *symbol* (word) address.

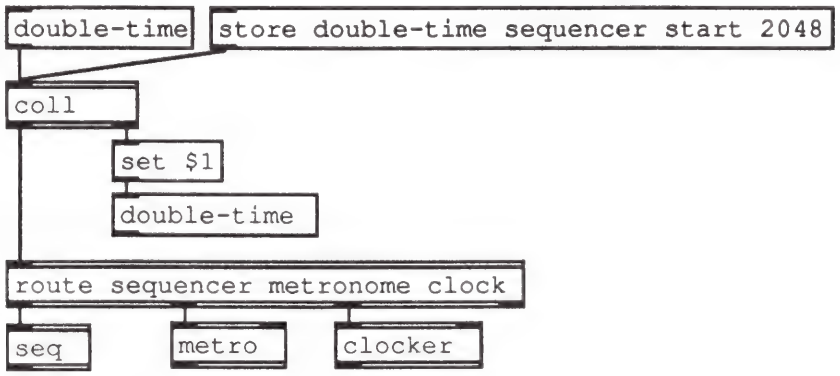
Any time **coll** receives a *list* in its inlet it uses the first number in the list as an address, and stores the *remaining* items in the list at that address. (You'll recall that a list is any space-separated set of items, beginning with a number.) For example, when **coll** receives the message 36 sequencer start 2048, it stores the message sequencer start 2048 at address 36. After that, whenever **coll** receives the number 36 alone, it sends the address (36) out the right outlet, and sends the message sequencer start 2048 out the left outlet.



You can also store messages with a *symbol* as an address instead of a number. If you just send it a message beginning with a symbol, **coll** will try to interpret the symbol as another kind of command, and won't store the rest of the message. So, to store messages with a symbol as the address, you must precede the symbol with the word store.

When **coll** receives a message beginning with the word store, it uses the first item after the word store as its address, and stores the rest of the message at that address. When **coll** receives that address alone in the inlet, it sends it out the right outlet (preceded by the word symbol), and sends the stored message out the left outlet.

Here's the same patch, using a *symbol* as an address for the message stored in **coll**, instead of a number.



coll precedes the symbol address it sends out its right outlet with the word symbol so that the address will not be interpreted as a command by other objects. For example, a message box will not be triggered by a word, because it will try to understand the word as some kind of command. However, if the word is preceded by symbol, the message box will be triggered and the word will replace a $1 changeable argument in the box.

Editing the Contents of coll

To view and edit the contents of a **coll**, double-click on the object and a Text window will open. If you make any changes to the Text window, you will be asked whether you want to keep those changes in the **coll** when you close the Text window.

The contents of a **coll** are written in a specific format. For details, look up **coll** in the *Objects* section.

Saving the Contents of coll

Once you have stored messages in **coll**, you can set it to save its contents as part of the patch. You unlock the Patcher window, select the **coll** object, choose **Get Info...** from the Max menu, and check the *Save coll with Patcher* option in the dialog box.

Alternatively, you can save the contents of the **coll** as a separate file (so the contents can be used by more than one patch). To do this, open **coll**'s Text window and choose **Save As...** from the File menu. Another way to save the contents as a separate file is to send a write message to the **coll** object, which opens a **Save As...** dialog box.

To *load* a file into a **coll** object, type the name of the file in as an argument, or send **coll** a read message, which will cause the **Open...** dialog to appear.

Tutorial 37

Data structures

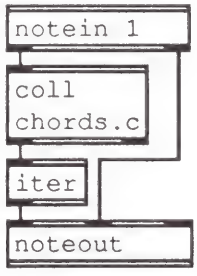
Storing Chords in coll

- Double-click on the **patcher** coll_examples object to open the subpatch window.
- Play some long notes on your MIDI keyboard. Every key on the keyboard has a unique 3-note chord assigned to it.

The chords are stored in a **coll** object, using the key number (the pitch of the played note) as the address.

- Double click on the **coll** object in Patch 1 to see how the chords are stored. If you want to change some of the chords, edit the numbers in the Text window, then close the window to update the contents of the **coll**.

When a pitch value is received in the inlet, **coll** sends out the 3-item list stored at that address. The list is broken up into a series of numbers by **iter**, and the numbers are sent (virtually simultaneously) to **noteout**, where they are combined with the note-on or note-off velocity being played on a MIDI keyboard.

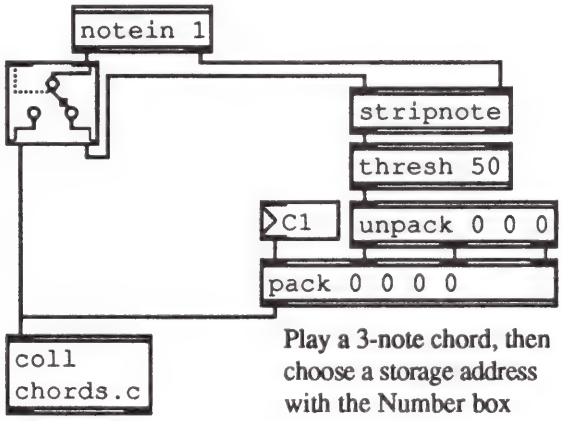


The rest of Patch 1 is for storing your own chords in a **coll**. When pitch data is routed out the right outlet of the **Ggate**, the note-on pitches are sent to a **thresh** object.

The **thresh** object is like **iter** in reverse. Numbers which are received within a certain *threshold* of time are packed together in order. The threshold is the maximum number of milliseconds between any given number and the previous one. When no new number is received within a certain period, the numbers are sent out as a list.

So, when you play notes of a chord simultaneously (within 50ms of each other) they are packed as a list, and after 50ms they are sent out. The **unpack** object selects the first three numbers and stores them in **pack**. Then, when you select an address by entering a number in the **Number box**, the address and the accompanying chord notes are all sent to **coll** as a list for storage. The **Number box** has been set to send only on mouse-up so that you can use it as a slider to enter an address. Otherwise, the chord would be stored in the address of every number you dragged through.

Tutorial 37



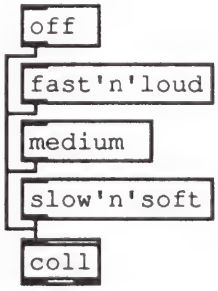
- Click on the **Ggate** to point its arrow to the right outlet.
- Play a 3-note chord that you want to store in **coll**. Play it as a 4-note chord first, to hear it along with the address note that will eventually trigger it, then play it as a 3-note chord to store it in **pack**.
- Use the **Number box** to select the address where you want to store your chord.
- Repeat the above steps until you have stored all the chords you want, then click on **Ggate** again to direct the played pitches back to **coll**. Play the address notes to hear the results.

If you want to save your new chords, you must open **coll**'s Text window again and choose Save As... from the File menu.

Parsing the Data Structures in a coll

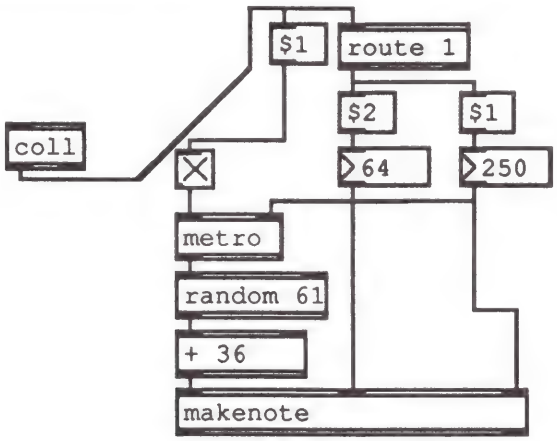
Patch 2 shows how **coll** can be used to store messages with symbol addresses, and it also shows how complex messages can be stored in **coll** and then parsed when they are sent out.

- Click on the different messages in Patch 2.



- Double-click on the **coll** object in Patch 2 to view its contents.

The format we have chosen for our data structure in this **coll** is: *metro command, tempo/duration,* and *note velocity*. Because each message stores the data in the same order, we can access individual items in the data structure and use the item in a specific way.



The data structure is parsed as it comes out of the **coll**. The message is first sent to **route**. If the first item of the message is 1 (meaning *metro on*), we use the remaining items in the message to supply a duration to **makenote**, a tempo to **metro**, and a velocity to **makenote**. If the first item in the message is not 1 (in this case, 0 is the only other possibility), nothing needs to be sent, so **route** ignores the message. After the essantial data is supplied to **metro** and **makenote**, the first item is used to turn the **metro** on or off.

Other Features of coll

These examples should give you a taste of what **coll** can be used for. There many other command messages which **coll** understands, too numerous to cover in detail in this Tutorial. For example, you can step through the different messages in **coll** with goto, next, and prev commands. And you can select or alter individual items of stored messages with commands such as nth (to get the *nth* item within a message), sub (to substitute an item in a messsage), and merge (to append items at the end of a message). For details about these commands, look under **coll** in the *Objects* section.

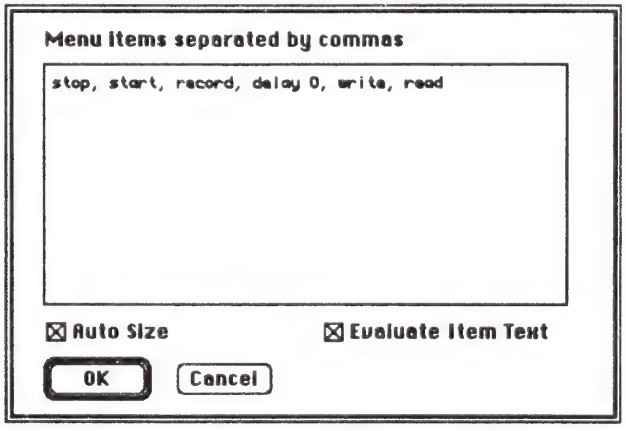
- Before you go on and look at the other subpatches, you will want to disable the chord-playing patch in the *[coll_examples]* window. Point the arrow of **Ggate** to the right outlet, or disable MIDI in the window by clicking on the MIDI enable/disable icon in the title bar. Close the *[coll_examples]* subpatch window.

menu

The **menu** object creates a pop-up menu in a Patcher window. It can be used to choose commands with the mouse, just like any other Macintosh menu, and it can also be used to display messages when the number of a **menu** item is received in the inlet.

When an item in the **menu** is selected with the mouse (or by a number received in the inlet), the number of the **menu** item is sent out the left outlet. The items in the **menu** are numbered beginning with 0.

A **Get Info...** dialog box appears when you first create a new **menu** object in the Patcher window. You type the **menu** items (commands) into the dialog box, separating them by commas. The **menu** commands can be any type of message: numbers, lists, words, sentences, whatever. If you want to include a comma *within* a **menu** command, you must precede it with a backslash (\).

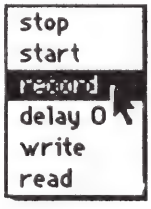


If you check *Evaluate Item Text* in the dialog box, **menu** will send the command out the right outlet verbatim before it sends the item number out the left outlet. If you check the *Auto Size* option, the width of the **menu** will automatically adjust according to the length of the text in **menu** commands.

- Double click on the **patcher** menu_examples object to open the subpatch window.

We've hidden many of the objects in this subpatch, to give you a visual idea of how **menus** may be used to enhance the user interface of a Patcher program.

- Record and play back a MIDI sequence using the *Sequencer* **menu** in the left part of the window.

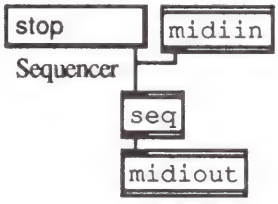


Tutorial 37

Data structures

It's easier to use a single **menu** than it is to click on a bunch of message boxes, it's more aesthetically pleasing, and it has the advantage of displaying the most recent command.

- Unlock the *[menu_examples]* window to see how the **menu** is connected in the patch.



The right outlet sends the actual text messages to **seq**.

In the other patch, the **menu** has a dual purpose of sending values and displaying the values it receives. If you have a lot of different sounds available on your synth, you may not be able to memorize all the program change numbers. A **menu** can help you associate the name of a sound (the text of a **menu** item) with a program change value (the item number).

When you select a **menu** item with the mouse, the item number is sent to **pgmout** as a program change value. Just as the names of sounds are specific to a given synthesizer, so may be the numbering system used by the synthesizer manufacturer. You'll need to figure out exactly how MIDI program change values correspond to the sound numbers on your synth. In this example we left menu item 0 empty, and used menu items 1 to 32 to store the names of sounds, so selecting a sound will transmit a program change value from 1 to 32.

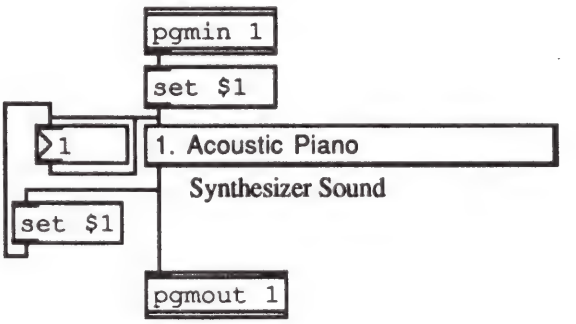
- Look at the **Get Info...** dialog for the *Synthesizer Sound* **menu**. Notice how we left the first **menu** item empty (by starting the text with a comma) so that we could use items 1 to 32.

How did we get more items in the **menu** than will fit into the dialog box? We typed the items in a Text window, then copied them and pasted them into the **Get Info...** box. You use the cmd-V equivalent for **Paste** since you can't access Max's menus in a dialog box.

- You may want to replace our list of sounds with one that corresponds to your equipment.

We also wanted to display *incoming* program change values, so we directed them to the inlet of our **menu**. However, we don't want the program change to be sent *out* again, so we use the set message to set the **menu** to the specified item without causing any output. Likewise, we want the **Number box** to reflect numbers from the **menu**, but we don't want it to send the number *back* to the **menu** because that would cause a *stack overflow*. Once again, the set message is the solution.

Tutorial 37



- Close the *[menu_examples]* window and double-click on the **patcher** preset_examples object to open the subpatch window.

preset

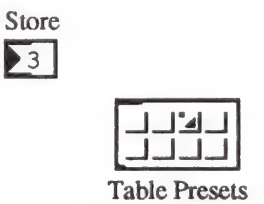
The **preset** object can store and recall the settings of other user interface objects in the same window—**sliders**, **dials**, **Number boxes**, **toggles**, etc. When you recall a stored setting, **preset** restores all these objects back the way they were at the moment the settings were stored. You can even connect the outlet of a **preset** to a **table** to store and recall various versions of the **table**'s contents.

The **preset** can operate in one of two ways. If the outlet of **preset** is connected to the inlet of other objects, it stores and recalls the settings of *only* those objects. If the outlet of **preset** is not connected to anything, it stores the settings of *every* user interface object in the window (except **tables**, which can *only* be stored by being connected to the outlet of a **preset**).

In the *[preset_examples]* window, the **preset** object is actually connected to the **table** with a patch cord, but we have hidden the patch cord for aesthetic reasons.

- Before you use the patch, enable **All Windows Active** in the Options menu. Then double-click on the **table** object to open its graphic editing window. You can draw in pitch values from 0 to 60 (which will be transposed up into the keyboard range by the *Offset* of the lower **hslider**), and then play those pitches by dragging on the upper **hslider**.

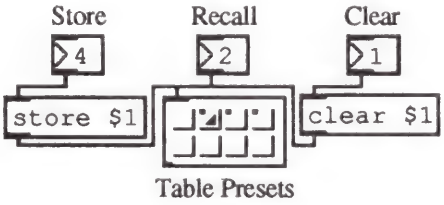
- When you have drawn a pitch curve that you like in the Table window, enter a number in the **Number box** marked *Store*, and the **table** values will be stored in that **preset** location.



- Repeat the process until you have stored several **table** presets. Then you can recall different ones by entering a number with the **Number box** marked *Recall*.

- Now unlock the *[preset_examples]* window to see what's going on behind the scenes.

The **Number boxes** labeled *Store*, *Recall*, and *Clear* are actually sending messages to the **preset**.



To store settings in a preset location, you send the message store, followed by the number of the preset location. To recall a preset, just send the number of that preset alone. To clear a preset, send the message clear, followed by the preset number. clearall will clear all stored presets.

You can save the contents of **preset** in a separate file with the write message, and load a file in with the read message.

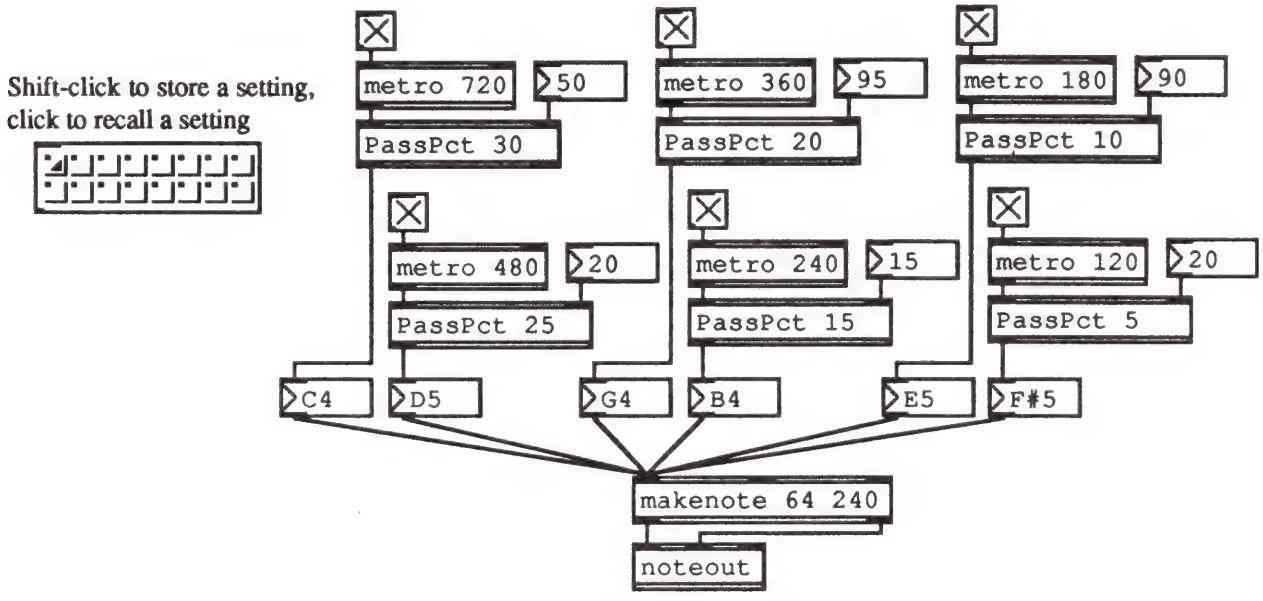
- Double-click on the **patcher** another_example object to see the sub-subpatch.

The **preset** object in the [another_example] window already has 16 presets stored in it, as part of the patch. To store the contents of a **preset** along with a patch, rather than as a separate file, you select the **preset** object, choose **Get Info...** from the Max menu, and check *Save Presets with Patcher*.

This patch shows you another way to store and recall presets: you Shift-click on a preset to store the current settings, and you can then recall the settings by just clicking on that preset. The number of preset locations in a **preset** object is not dependent on the object's physical size. Each **preset** object holds 256 preset locations, even if they aren't shown within its object box.

- Click on different preset buttons to recall different **toggle** and **Number box** settings. Try creating your own repeated note patterns and storing your settings in the **preset** object.

Tutorial 37



There are no hidden patch cords in this window. When a **preset** is not connected to anything, it stores the setting of *every* user interface object in the window.

Summary

A *data structure* is used to store data so that individual items can be easily accessed.

The **coll** object stores any kind of message, with either a number or a symbol as the address. Data to be stored can be received as messages in the inlet or typed into **coll**'s text editor window. When **coll** receives an address in its inlet, it sends the address out its right outlet, and sends the message stored at that address out its left outlet.

The contents of a **coll** object can be stored as part of the patch that contains it, or as a separate file. A **coll** file can be loaded into a **coll** object with the read message, or by typing the file name in as an argument.

The message sent out by **coll** can be parsed by other objects to select particular items from the data structure. Also, individual data items can be sent out or altered by certain commands in **coll**'s inlet.

The **menu** object is a pop-up menu in a Patcher window, and the **menu** items (commands) can be any kind of message. The **menu** may be used for selecting commands with the mouse and/or for displaying messages. When a **menu** command is selected, either with the mouse or by a **menu** item number received in the inlet, **menu** displays the command, (optionally) sends the stored message out the right outlet, and (always) sends the item number out the left outlet.

The **preset** object lets you store the settings of every other user interface object in the window at a certain point in time, then recall those settings at some later time. If **preset** is connected to certain objects with patch cords, it stores and recalls the settings of *only* those objects. The contents of a

table can also be remembered by **preset**, but the **table** *must* be connected to **preset**. The **preset** object can store and recall up to 256 different collections of the settings of all user interface objects.

See Also

coll	Store and edit a collection of different messages
Data Structures	Ways of storing data in Max
menu	Pop-up menu, to display and send commands
preset	Store and recall the settings of other objects

expr
and **if**

Tutorial 38

C Language Expressions

The Max application itself is written in the C programming language, and many of the terms and object names (such as **&&** and **||** for *and* and *or*) in Max have a basis in C. For programmers who have some experience with C or Pascal, and who feel comfortable using traditional programming language syntax, Max provides objects for evaluating mathematical expressions and conditional statements which are expressed in a C-like way.

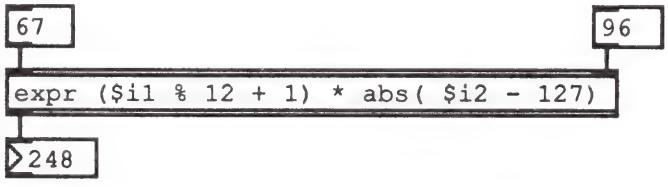
Even if you don't know a programming language, you can understand and use these objects. Often a complex comparison or mathematical calculation that would require several Patcher objects can be expressed in a single phrase, in a single object. Also, you can do a few calculations with these objects that you can't do with any of the other arithmetic operators.

expr

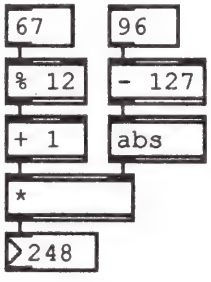
The arguments to the **expr** object make up a mathematical expression (formula) in a format that is similar to C programming language. For example, the C expressions...

```
x = 67; y = 96; z = (x % 12 +1) * abs(y - 127);
```

can be expressed in an **expr** object as



Without using **expr**, you would perform the calculation with a patch with many objects that looks like this:

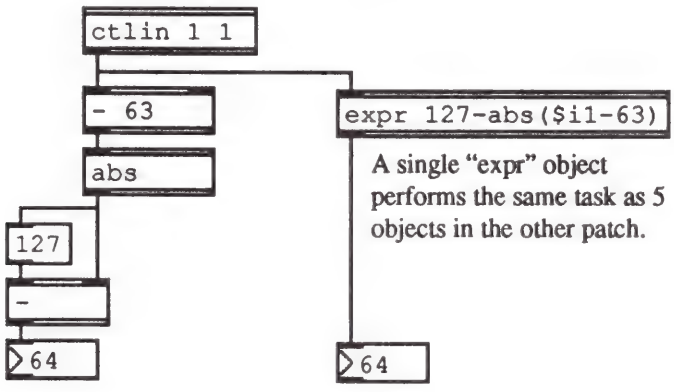


- To see an example of an object-oriented solution to a programming problem, and a comparable solution using **expr**, double-click on the **patcher** expr_example object.

Tutorial 38

expr
***and* if**

For this example, we want to solve the following problem: as the modulation wheel progresses from 0 to 127, send pitch bend values from 64 to 127 and back down to 64. The patch on the left shows a standard Patcher way of doing this. The patch on the right shows the different tasks all combined into a single mathematical expression in **expr**.



Notice that the changeable arguments in an **expr** object include a letter, as in $i1, to tell **expr** what data type to use for that argument (i for int, f for float).

- Move the modulation wheel on your MIDI keyboard from 0 to 127, and you'll see that both methods of stating the mathematical expression work equally well. However, it's a bit more memory-efficient to use a single object instead of five.

if

Another staple of C programming is the *if ( ) ; else ;* combination. In Pascal, this is expressed as *IF condition THEN statement ELSE statement.* In plain English this means: if a certain condition is met, do one thing, otherwise do another thing. Sometimes this way of thinking about the world just seems to make a lot more sense than a bunch of boxes connected together with wires!

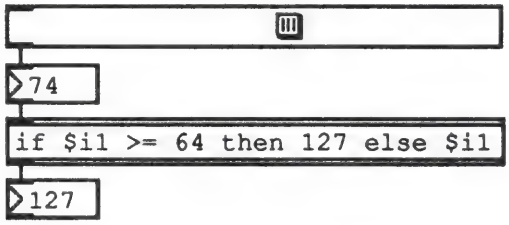
Max has an object called **if** which lets you express programming problems in an *if/then/else* format. If the comparison in the arguments is true (does not equal 0), then the message after the word then is sent out the outlet, otherwise, the message after the (optional) word else is sent out.

So, the conditional statement

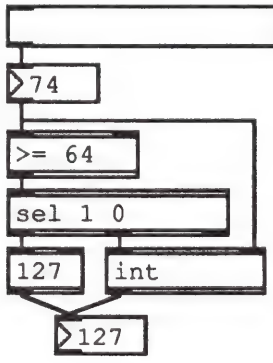
if the received number is greater than or equal to 64,
send out 127,
otherwise send out the received number

Tutorial 38

would be expressed as

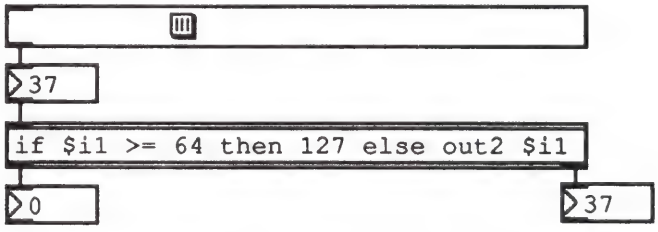


An object-oriented way of saying the same thing might be:



The then and else portions of the **if** object contain a message similar to that you would type into a message box. You can include changeable arguments, but not mathematical expressions as you can in the portion of the message after the if.

If the then or else portions of the **if** object begins with the argument out2, then the object has a second outlet on the right, and the message is sent out the right outlet.



The then portion and else portions can also begin with send, followed by the name of a **receive** object. In that case, the output is sent to all **receive** objects with that name, instead of out the outlet.

- Double-click on the **patcher** if_example object to see the usefulness of **if**.

The problem in this example was, "if the note G1 is played with a velocity between 16 and 95, start a sequence, otherwise increment a **counter** somewhere else." The example shows that a great many tasks can be combined into a single *if/then/else* expression. In this instance, one **if** object does the work of nine other objects.

Tutorial 38

expr
and **if**

C Math Functions

The C math library has many functions for such calculations as logarithms, trigonometric ratios, *x* to the power of *y*, and so on. Max does not have specific objects for these functions, but they can be included in the arguments of an **expr** object. This is a real strength of **expr**, because it lets you make calculations you would not otherwise be able to make in a Max program.

In the main patch of this example, we use two different math functions, sin() and pow() to calculate pitch bend curves to be stored in the **table**. One formula makes a sine wave with a range of ±63 and an offset of 64. The other formula draws exponential curves from 64 to 127.

- Check the **All Windows Active** option and double-click on the **table** object to open its graphic window. Click on the **button** at the top of the patch to draw a sine wave in the Table window.

The expression in **expr** converts the input to a float by using the $f1 argument (instead of $i1), in order to do a floating point calculation. It divides the input by 128, (so as the input progresses from 0 to 127 it will produce a progression from 0 to almost 1), multiplies the input by 2π (approximately 6.2832), and calculates the sine of that amount. The resulting sine wave values are multiplied by 63 and offset by 64.

```
expr sin($f1 * 6.2832 / 128.) * 63. + 64
```

- The expression in the other **expr** is a simple exponential mapping function. Click on **Ggate** to point it to the right outlet. Drag on the **Number box** to select an exponent for the curve to be calculated by **expr**. An exponent of 1 produces a straight line, an exponent greater than 1 yields an exponential curve, and an exponent less than 1 yields an inverse exponential curve.

- Click on the **button** to draw the curve in the Table window. Try different exponent values and redraw the curve.

You have probably noticed that the curves are not drawn in the Table window in the 127 milliseconds specified by the **line** object. This is partly due to the time it takes to draw on the screen, but it's also because exponential calculations—especially with a large exponent—take a longer time for the computer to calculate. For this reason it's better to perform such calculations in advance and store the values in a **table** to be accessed later, rather than to calculate the values on the fly.

Once the curve is stored in the **table**, it is read through by another **line** object each time you play a note on your keyboard. The values are sent to **bendout** to be transmitted to the synth as pitch bend. The speed with which **line** reads through the **table** depends on the velocity of the note you play.

- Play long notes on your keyboard with widely varying velocities, and listen to the different speeds with which **line** reads through the curve in the **table**. Draw different curves in the **table** to hear their sonic effect.

expr
and **if**

Tutorial 38

Summary

The **expr** object takes a C-like mathematical expression as its argument, including changeable arguments. When a number is received in the left inlet, **expr** replaces the changeable arguments, evaluates the expression, and sends out the result.

The **if** object evaluates a conditional statement in the form "**if** *x* is true then output *y* else output *z*". The conditional statement can contain changeable arguments. The output can be sent to **receive** objects instead of out the outlet.

Both **if** and **expr** are capable of combining the functionality of several Patcher objects into a single object, which is usually more memory-efficient.

The expression in the argument of **expr** can contain C math functions such as pow() and sin(), and can also contain relational operators. For details on the operators and functions you can use, look under **expr** in the *Objects* section.

See Also

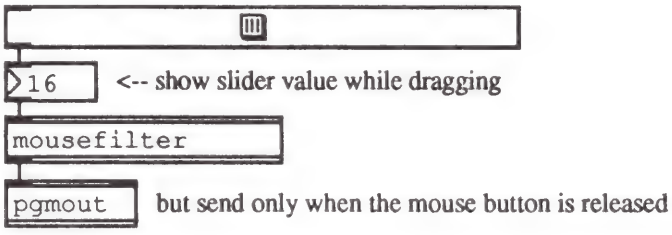
expr	Evaluate a mathematical expression
if	Conditional statement in if/then/else form

Tutorial 39

Mouse control

mousefilter

There may be times when you want to see the exact value that is going to be sent out of a **slider** or **dial** *before* it is actually sent. The **mousefilter** object helps you do that. It receives numbers in its inlet, but passes them on only when the mouse button is up. Consider the example below.



While you are dragging on the **hslider**, the numbers are sent to the **Number box** for display, but **mousefilter** does not pass them on because the mouse button is down. When you release the mouse button, the last number is sent out the outlet of **mousefilter**.

The patch in the left part of the Patcher window is very similar to this example, except that we have hidden the **mousefilter** object and the patch cords. When you drag on the **dial**, the upper **Number box** shows the output of **dial**, but no number is sent to the lower **Number box** (and to **pgmout**, also hidden) until the mouse button is released.

- Drag on the **dial** to select an new program change number. Nothing is sent to your synth until you release the mouse button.

The **button** at the top-left corner of the patch triggers a **line** object to step through the program changes automatically over the course of 16 seconds.

- Click on the **button**. While the **dial** is being automatically controlled, you can use the mouse to suppress certain program change numbers. Whenever you hold the mouse button down, **mousefilter** acts as a gate to shut off the flow of numbers to **pgmout**.

- Unlock the Patcher window to see how the connections are made. Lock the window again before trying the rest of the patch.

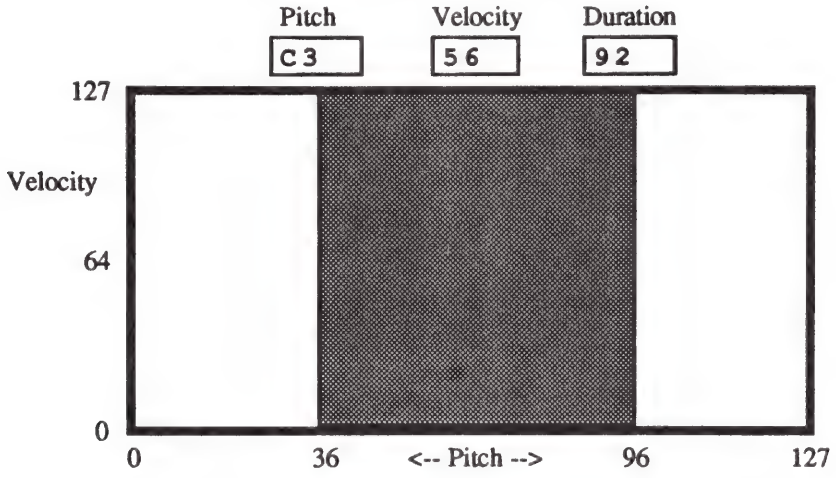
Using the Mouse Position to Provide Values

The large box in the example Patcher window was imported from a graphics application and pasted into the window using **Paste Picture**. It delineates a pitch-velocity grid in which you can drag with the mouse to play notes. The gray area shows the pitch space which corresponds to the range of a 61-key keyboard (C1 to C6).

- Before using the pitch-velocity grid, you must click on the most extreme bottom-left corner of the box. This tells Max where the 0,0 point is.

- After you have done that, click and/or drag inside the box to play notes with the mouse.

Tutorial 39



When you hold down the mouse button inside the box, notes are played continuously. Moving the mouse from left to right in the box increases the pitch. Moving from bottom to top increases the velocity. Large changes upward or downward cause the tempo of the notes to increase or decrease.

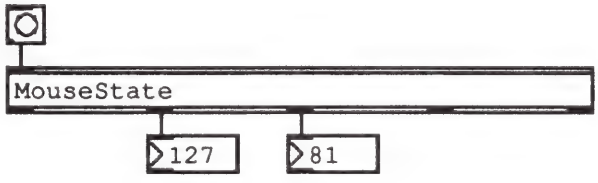
The mouse is not moving any kind of slider or other type of user interface object, so how does it send out notes?

- Unlock the Patcher window and scroll to the right to see what's going on.

MouseState

The generator of numbers in this patch is the **MouseState** object. When it receives a bang in its inlet, **MouseState** report the current horizontal and vertical location of the mouse out its left-middle and middle outlets.

A location on the screen is expressed as a horizontal-vertical pair of numbers, normally measured as the number of pixels away from the upper-left corner of the screen. Horizontal location is measured from left to right, and vertical location is measure from top to bottom.

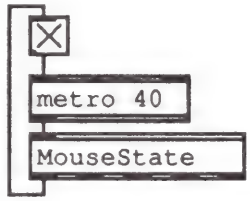


The cursor is 127 pixels to the right of, and 81 pixels down from, the upper-left corner of the screen

When **MouseState** receives the message zero, it uses the current mouse location as the new 0, 0 point, and makes all subsequent measurements in terms of that new point. That's why you click on the bottom-left corner of the pitch-velocity box before starting. We've situated a tiny **ubutton** object

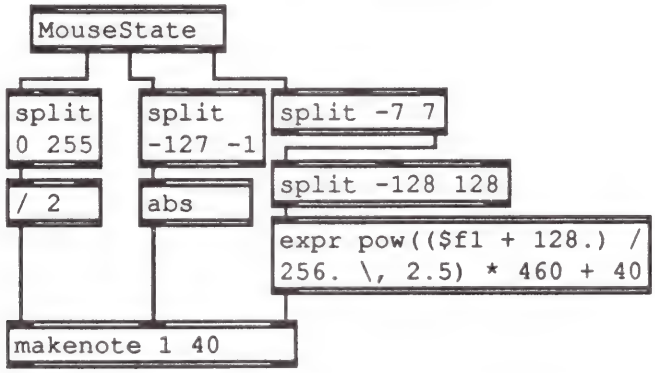
on the corner of the box, so when you click there it triggers a zero message to **MouseState** and sets that point as the new 0, 0 point.

MouseState also reports the status of the mouse button. It sends out 1 when the button is pressed and 0 when it is released. We use this feature in our patch to control the **metro** object which sends bangs to **MouseState**. Since **metro** only sends a bang when the mouse button is down, **MouseState** only sends out location values while the mouse is down.



The two right outlets report the *change* in location since the previous report. The horizontal location, the vertical location, and the change in vertical location are used in this patch to supply values for pitch, velocity, and tempo. Each range of values has to be limited and processed slightly differently to place the values in an appropriate range.

For example, the pitch-velocity box is 256 pixels wide, so we limit the horizontal location values between 0 and 255 with a **split** object, then divide them by 2 to get a range of pitches from 0 to 127. The box is 128 pixels high, but remember that vertical location is measured from top to bottom, so when the mouse is in the box the vertical values will range from 0 to -127. We therefore limit the vertical values between -127 and -1 and use the **abs** object to make the values positive. (We don't want any 0 velocities because they'll be supplied by **makenote**.)



To get the tempo, we use the change in vertical location of the mouse. But we only want to detect *substantial* change, so we first filter out slight changes (±7 pixels). Then we limit the values between -128 and 128 and use **expr** to map that range onto an exponential curve from 40 to 500. Thus, a large increase in velocity causes a fast tempo, while a large decrease in velocity causes a slower tempo. A gradual change in velocity does not change the tempo.

Mouse control

Tutorial 39

Summary

When the mouse button is down, **mousefilter** suppresses all numbers it receives until the button is released, then it sends out the last number it received. **mousefilter** can be used as a mouse-dependent gate, especially to allow you to view many numbers but only send on the ones you want.

Every time **MouseState** receives a bang, it sends out the location of the mouse and the *change* in location since the last report. These values can be used to provide continuous musical control, giving you the ability to use the entire screen as as a field in which to produce values in two dimensions by moving the mouse.

When the mouse button is pressed **MouseState** sends 1 out its left outlet, and when it is released **MouseState** sends out 0. These values can be used to turn a process on and off, or to open and shut a **gate**.

See Also

mousefilter	Pass numbers only when the mouse button is up
MouseState	Report the status and location of the mouse

Tutorial 40

Automatic actions

Opening a Subpatch Window

Your programs can automatically open and close Patcher windows and detect when a window is opened or closed, triggering some action.

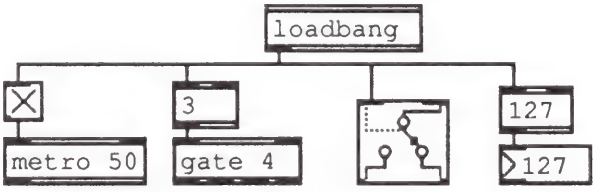
- When you open the example patch for this chapter of the Tutorial, the window of the subpatch object **stopwatch** is opened immediately and begins to display the time elapsed since the window was opened.

As we explained earlier in Tutorial 26), a **patcher** object will open automatically if you leave its subpatch window open when you save the Patcher that contains it. A subpatch saved as a *separate file*, however, (such as **stopwatch**) will always have its window closed when the main patch is opened.

To open the *[stopwatch]* window, which would normally not be open, we used two objects, **loadbang** and **pcontrol**.

loadbang and closebang

The **loadbang** object sends out a bang once when the Patcher that contains it is opened (loaded into memory). This allows you to trigger certain actions *immediately* when a Patcher is loaded. You can use **loadbang** to open **gates** and **switches** (which are closed when a patch is opened), start timing objects such as **metro**, or supply initial number values to an object such as **Number box**.



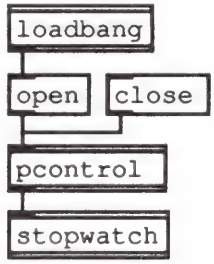
The counterpart to **loadbang** is **closebang** (not shown here) which can be used to trigger actions—such as turning off a **metro**—when a Patcher is closed.

pcontrol

Subpatch windows can be opened and closed by the **pcontrol** object. When **pcontrol** receives an open or close message in its inlet, it opens or closes the window of any subpatch objects connected to its outlet.

In the left part of the main Patcher window you can see how the *[stopwatch]* window was opened automatically. The bang from **loadbang** triggered an open message to **pcontrol**, which opened the window of the **stopwatch** object.

Tutorial 40



Using **pcontrol**, you can produce multi-window patches with each window displaying something different, and you can make **pcontrol** show or hide windows when appropriate.

Note: Because opening and closing windows takes some time, it's not advisable to do it while Max is playing music, unless you're in **Overdrive** mode.

- You can stop and restart the **stopwatch** by clicking on the **toggle** in the *[stopwatch]* window. To open and close the *[stopwatch]* window, send open and close messages to **pcontrol**.

- Close the *[stopwatch]* window, and open the *[clicktrack]* window by sending an open message to the other **pcontrol** object. When the *[clicktrack]* window is opened, a 4-note click track automatically begins to play, at the metronomic tempo shown in the **Number box**.

The **pcontrol** object can also enable and disable MIDI objects in the subpatch windows it controls. The message enable 0 in the inlet of **pcontrol** disables the MIDI objects in the subpatch, and enable 1 (or any number other than 0) re-enables them. Bear in mind, if you make a patch that automatically disables the MIDI objects in a subpatch, that you run the risk of causing stuck notes on the synth if you cause a note-off message to be lost.

- Enable **All Windows Active** so that you can click in the main Patcher window without bringing it to the foreground. Then click on the toggle in the main Patcher window to toggle the MIDI Enable/Disable button in the title bar of the *[clicktrack]* window. The sound stops, but the **led** continues to flash.

led

The **led** object is an *on/off* indicator similar to **toggle**, but not identical. Whereas **toggle** passes on any number it receives, **led** outputs only 0 or 1 indicating the zero/non-zero status of the number it receives. When **led** receives a bang, it flashes and outputs 0. You can change the color and flash time of an **led** object by selecting it and choosing **Get Info...** from the Max menu.

active

What makes the **clicktrack** and **stopwatch** objects run automatically when their windows are brought to the foreground? They are controlled by another automatic control object, **active**.

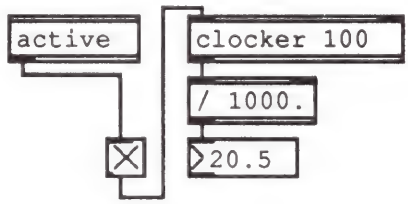
- To see the hidden objects in the subpatches, you must open the actual file in which the subpatch is saved. Choose **Open...** from the File menu to open the file named *stopwatch*.

Tutorial 40

Automatic actions

- Stop the time display by clicking on the **toggle** in the *stopwatch* window, then unlock the window to see its hidden objects.

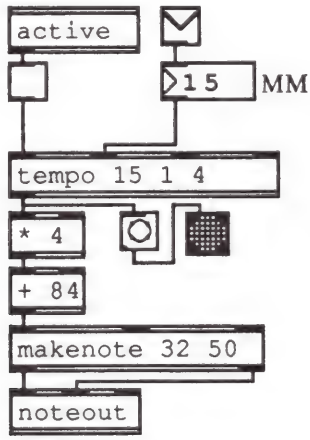
When a window is made active (i.e., brought to the front), the **active** object in that window sends out 1. When the window made inactive (is no longer in front) **active** sends out 0. We have used **active** to turn a **clocker** object on automatically whenever the window is brought to the foreground.



The **active** object sends out a number *only* in response to a change in its foreground/background status, and is not affected by the setting of **All Windows Active**. When **All Windows Active** is checked, you can click in any window without first bringing it to the foreground, but only the foreground window is technically *active*. When you move a window to the background, an **active** object in that window sends out 0, but when you close the window **active** does not send a 0, because it's not actually being sent to the background.

Even though the **stopwatch** object doesn't get any messages from other objects, it needs to have an inlet so that it can be controlled with **pcontrol**. You can include a *dummy* **inlet** object in a patch for this purpose.

- Close the *stopwatch* patch and open the file named *clicktrack*. Turn off the **toggle** in the *clicktrack* window, and unlock the window to see its hidden objects. You can see that it contains an **active** object to turn on **tempo** whenever the window is made active. The numbers 0 to 3 sent out by **tempo** are multiplied and transposed to play the pitches C5, E5, G#5, and C6.



Tutorial 40

The **clicktrack** object has one inlet for receiving new tempo values, but this same inlet can be used by **pcontrol** in the main patch to control the *[clicktrack]* window.

Summary

The **loadbang** object sends a bang whenever the patch that contains it is loaded into memory. The **closebang** object sends a bang when the patch that contains it is closed. These bangs can be used to start processes, open or close a **gate**, send a message, etc.

When the **pcontrol** object receives an open or close message, it opens or closes all subpatch objects connected to its outlet. The MIDI Enable/Disable button of a subpatch window can also be toggled on and off by **pcontrol** with the commands enable 1 and enable 0. Disabling MIDI objects while a note is being played, however, may cause a note-off message to be lost, leaving a stuck note on the synth.

The **active** object sends out 1 when the window that contains it is brought to the front and 0 when some other window is brought to the front.

See Also

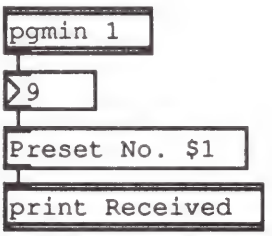
active	Send 1 when window is active, 0 when window is inactive
closebang	Send a bang automatically when patch is closed
loadbang	Send a bang automatically when patch is loaded
pcontrol	Open and close subwindows within a Patcher

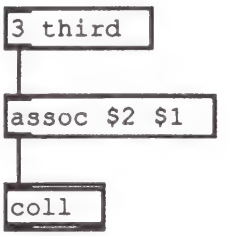
Arguments

$ and #, changeable arguments to objects

$ in a message box

The dollar sign ($) is a special character which can be used in a message box to indicate a *changeable argument*. When the message box contains a $ and a number (such as $2) as one of its arguments, that argument will be replaced by the corresponding argument in the incoming message before the message box sends out its own message.

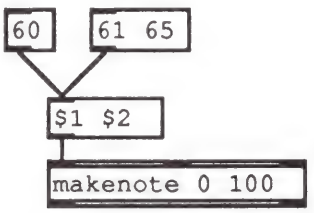


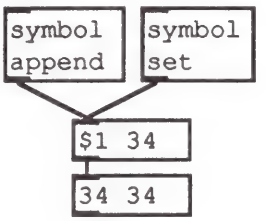


In the left example above, the $1 argument in the message box is replaced by the number received in the inlet (in this case 9) before the message is sent out. The message printed in the Max window will read Received: Preset No. 9.

The right example shows that both symbols and numbers can replace changeable arguments. It also shows that changeable arguments can be arranged in any order in the message box, making it a powerful tool for rearranging messages. In the example, the message assoc third 3 is sent to the **coll** object.

When a message box is triggered without receiving values for all of its changeable arguments (for instance, when it is triggered by a bang), it uses the most recently received values. The initial value of all changeable arguments is 0.





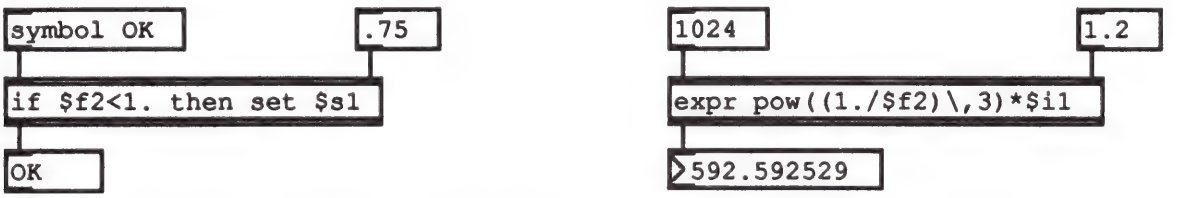
In the left example above, a message of 60 will initially send 60 0 to the **makenote** object. After the 61 65 message has been received, however, the number 65 will be stored in the $2 argument, so a message of 60 will send 60 65 to **makenote**.

A message box will not be triggered by a word received in its inlet (except for bang), unless the word is preceded by the word symbol. In such a case, the $1 argument will be replaced by the word, and not by symbol. In the right example, the $1 argument is replaced by either set or append, and the message set 34 or append 34 is sent to the next message box.

To include a dollar sign in a message without it having a special meaning, you must precede the $ with a backslash (\).

$ in an object box

A changeable $ argument can also be used in the **expr** and **if** objects. In these objects, the $ must be followed immediately by the letter i, f, or s, indicating whether the argument is to be replaced by an int, a float, or a symbol.



If the message received in the inlet does not match the type of the changeable argument (for example, if an int is received to replace a $f argument), the object will try to convert the input to the proper type. The object cannot convert symbols to numbers, however, so an error message will be printed if a symbol is received to replace a $i or $f argument.

in object and message boxes

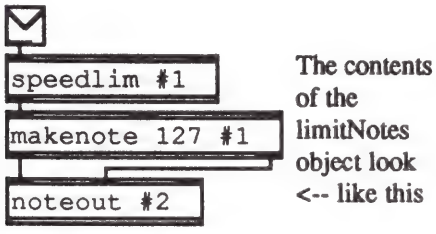
When you are editing a Patcher which will be used as a *subpatch* within another Patcher, message boxes and most object boxes in the subpatch can be given a changeable argument by typing in a pound sign and a number (for example, #1) as an argument. Then, when the subpatch is used inside another Patcher, an argument typed into the object box in the Patcher replaces the # argument *inside* the subpatch.

In this way, **patcher** objects and your own objects can require typed in arguments to supply them with information, just as many Max objects do. A symbol such as #1 is a *changeable argument*, and is replaced by whatever number or symbol you type in as the corresponding argument when you use the patch as an object inside another patch. A changeable argument cannot be used to supply the name of an object itself, but can be used as an argument anywhere inside your object.

In the following example, arguments typed into the **limitNotes** object boxes supply values to the objects inside **limitNotes**. When the **hslider** is moved, one **limitNotes** object plays a note every 300 milliseconds on MIDI channel 5, and the other plays a note every 200ms on MIDI channel 7.

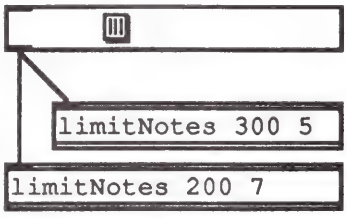
Arguments

$ and #, changeable arguments to objects



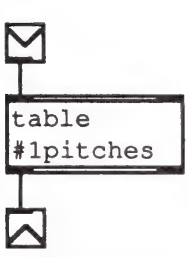
These are Max objects

When limitNotes is used as an object in another patch, #1 is replaced by the 1st argument, and #2 is replaced by the 2nd argument

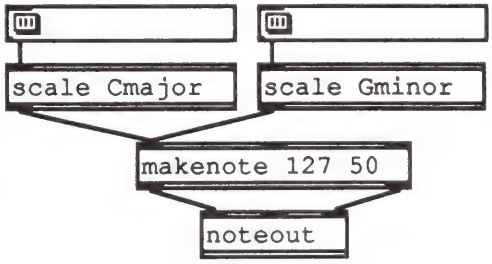


limitNotes is a patch saved as a document

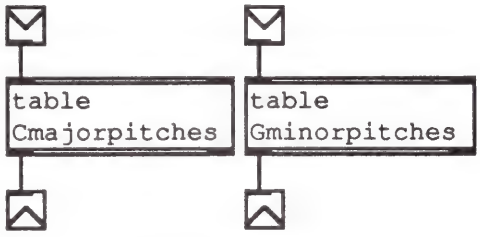
A pound sign and a number can even be *part of* a symbol argument, providing variations on a name. In the example below, the #1 part of the changeable argument inside **scale** is is replaced by the argument in the patch that uses **scale**. The **scale** objects will each use a different pre-saved **table** file, producing different results.



The contents of the "scale" object look like this

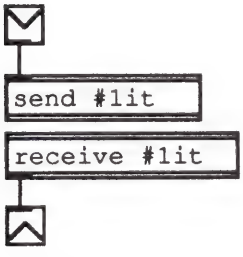


The argument is supplied in the patch that uses "scale"...

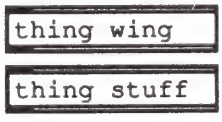


...and the effect is as if there were two "scale" objects that look like this

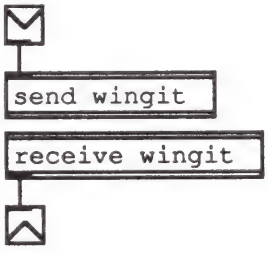
This technique is also useful for creating **send** and **receive** objects which are "local" to a subpatch, as shown in the example below.



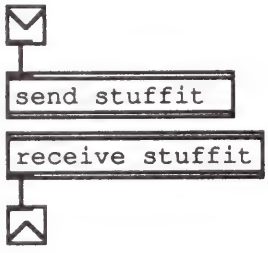
The contents of the "thing" patch using **send** *and* **receive**



Parent patcher with two "thing" objects with arguments



Contents of **thing** wing



Contents of **thing** stuff

Ways of storing data in Max

Data Structures

Storing Data

Max has objects specifically designed for storing and recalling information, ranging from simple objects which store a single number to more powerful objects for storing any combination of messages.

The simple **int** and **float** objects store a number and then output it in response to a bang. The **value** object allows a single value to be changed or recalled from different Patcher windows. The **accum** object stores a single number which can be added to or multiplied.

A *data structure* stores information together in a consistent format, so that a particular item can be retrieved using the address (location) of the item.

Arrays

The **table** object is an *array* of numbers, where each number in the table has a unique index number, or address. When an address is received in the left inlet, the value stored at that address is sent out the left outlet. Storing numbers in an easily accessible way is the main utility of such an array, but the **table** object has many powerful features for modifying and using the numbers it stores.

The values in a **table** are displayed graphically in the *table editing window*, showing the addresses on the *x* axis of a graph, and the values on the *y* axis. You can change the values displayed in the table window by drawing in the graph with drawing tools, or by cutting and pasting regions of the graph.

Other messages sent to a **table** can store new values, change its size, report the sum of all its values, step forward or backward through different addresses, report the address of a specific value, and provide statistical information about its values.

The **funbuff** object stores addresses and values, but unlike a **table**, the addresses can be any number, and gaps can exist between addresses. If **funbuff** receives in its inlet an address that does not actually exist (a number that falls in a gap between existing addresses), it finds the next smallest address, and outputs the value at that address.

The **bag** object stores a collection of numbers without any addresses. Numbers can be added to and deleted from the **bag**, and a bang in its inlet sends out all of the currently stored numbers.

Complex Data Structures

The **preset** object is also a kind of array, but each address in its array contains the settings of other objects in a Patcher window. When an address number is received in **preset**'s inlet (or when you click on one of **preset**'s buttons), the settings of those objects are changed to the values stored in the **preset**. In this way, every user interface object in the same window as the **preset** object has its settings stored and recalled. Alternatively, you can connect the outlet of a **preset** to *some* of the window's user interface objects, making them the only ones affected by that **preset**.

Data Structures

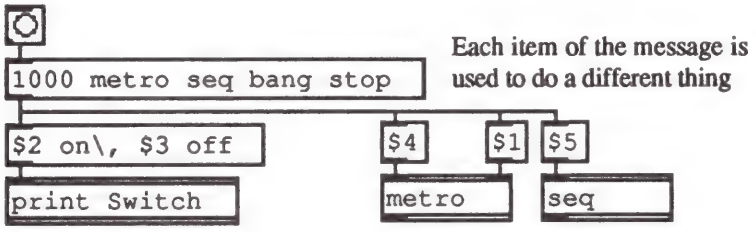
Ways of storing data in Max

The **coll** object (short for *collection*) stores numbers, symbols, and lists. A single address in **coll** can be either a number or a symbol. You can also modify stored data in a **coll** with messages such as sub, which changes a single item in a stored location, or merge, which appends additional data to a location. You can also access an individual item in a list stored in a coll with the nth message.

A **coll** object is useful for recording and playing back a "score" that has lists of times, pitches, and durations. Or you could use a **coll** to store a collection of text messages to be shown to the user when certain numbers or symbols are received.

You can edit the contents of a **coll** in a standard Max text editor window by double-clicking on its object box. The standard Max text editor window will open. The text format used is discussed in the description of the **coll** object.

The message box can be considered a kind of data structure, since it can hold up to 256 different items as arguments. The contents of a message can be modified using set and append messages, and message boxes can include changeable arguments which are replaced by the arguments of messages it receives in its inlet. Individual items in a message box can be accessed by sending its contents to *another* message box with changeable arguments, as shown in the example below.



The user interface object **menu** is essentially an array of symbols. When the number of a menu item is received in the inlet, the item text is displayed and can also sent out the right outlet if desired. Item text is changed with a setitem message. When a user chooses a menu item with the mouse, she is, in a way, specifying a symbol, causing the symbol's address is sent out the left outlet.

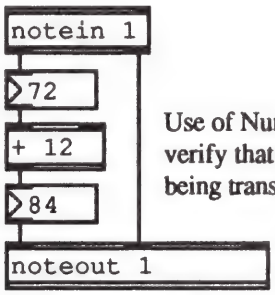
See Also

coll	Store and edit a collection of different messages
funbuff	Store x,y pairs of numbers together
menu	Pop-up menu, to display and send commands
message	Send any message
table	Store and graphically edit an array of numbers
Tables	Graphic editing window for creating **table** files

Viewing Messages

There are several good ways to see exactly what messages are actually passing through the patch cords of your program, so that you can be sure it's doing what you want. The best way to view messages is to include extra objects in your patch temporarily, which "intercept" the messages as they are sent.

For viewing numbers, the **Number box** can be used as a kind of "wiretap" in any patch cord. Numbers will pass through the **Number box** unchanged, but will also be displayed as they go through.

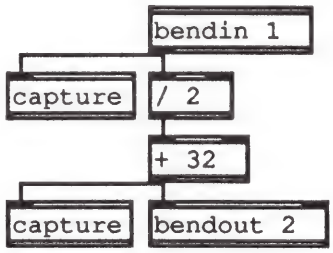


Use of Number boxes to verify that the pitch is being transposed properly

The **Number box** has several drawbacks as a debugging tool. It can only show a single int or float value, not a list of different values. Numbers may pass through it too fast for your eye to follow them. If the same number passes through several times in a row, you won't see any change in the display. And finally, there is now way to see previous numbers once a new number is displayed.

The **capture** object solves all of these problems by handling both numbers and lists of numbers, and by storing an arbitrary number of values at a time. Hook up a **capture** object off to the side at the point where you want to look at some numbers, then double-click on its object box to open a text editing window which displays the numbers that have recently been received. The default number of values **capture** holds is 512, but this size can adjusted with a typed-in argument.

You can double-click on the capture objects to see the last 512 numbers received at each point



Another advantage of **capture** is that you can copy numbers from its editing window and paste them into another file or into a **table**. Even though **capture** continues to receive numbers, they do not automatically appear in the editing window, so you have to re-open its editing window each time you want to view any newly received numbers.

To see *any* kind of message—symbols, numbers, lists, whatever—use **print**. The **print** object does not try to understand the messages it receives, it just posts them verbatim in the Max window. The Max window scrolls as each new message is printed, and you can scroll up to see previous

messages. The disadvantage of **print** is that the time needed to print the messages and scroll the Max window is often greater than the time between messages, so **print** may get behind, affecting the timing of your patch.

If all you need to do is verify that some message, any message, has been sent, use a **button**, which will flash each time it receives any kind of message.

Error Messages

If you make a programming error, Max will print an error message telling you about the mistake in the Max window. Many errors are reported while you are editing in the Patcher window, but other errors do not become evident until you actually run your program. A list of error messages, likely causes of each message, and possible solutions can be found in this section under *Errors*.

Test As You Go

It is infinitely easier to debug a small patch than it is to debug a big, complicated one. It is also much easier to debug a large, complicated patch when you know for sure that certain parts of it work correctly.

At every single stage in the development of a patch, test everything as you go along. Try sending extreme and unusual messages to your patch, as well as normal, expected ones, to make sure that your patch doesn't malfunction in situations you haven't considered. Once you are sure that a portion of your program works properly, you may want to encapsulate that portion by saving it in a separate file, and using it as a subpatch in a larger patch.

Comment

There is probably no known case of a programmer complaining because a program contains *too many* comments. Explanatory notes can help others understand your patch, and can help *you* remember what you have done, when you go back later to fix a bug you have discovered. It is surprising how fast you can forget why you wrote a program the way you did. You may even want to use a Text window to make notes to yourself to jot down ideas for future reference.

See Also

Efficiency	Issues of programming style
Encapsulation	How much should a patch do?
Errors	Explanation of error messages

Efficiency

Program Size and Speed

When you are writing very big, complicated patches, or are linking many subpatches together inside one main patch, matters of program size and computational efficiency come into play.

When you open a Patcher file, each object in the patch is loaded into the internal memory of the Macintosh. A very large patch containing many objects and subpatches can take up a considerable amount of memory as well as a long time to load. Therefore, you may wish to consider how to build patches that avoid superfluous messages and objects.

There are usually several ways to accomplish the same programming task in Max, and usually one way will be more efficient than another in terms of program size and speed.

There are three efficiency issues to consider:

1. The loaded size of a Max program is a function of the number of objects (and subpatches), and the complexity of each one.

2. The load time of a complex program also a function of the same two factors.

3. The "real-time" computational efficiency of a program is affected by the fact that some objects are more efficient than others in operation and communication.

Principles of Efficiency

Since there are so many different kinds of messages that can be sent in Max, an object often has to "look up" the meaning of the message it sends or receives. Computational speed is achieved primarily by avoiding this *message lookup*. Look at the description of the inlets and outlets of two connected objects in the Objects section to see if the share the same message type. In this case, Max will not have to do any "interpretation" of a message.

gate, **switch**, **Ggate**, and **Gswitch** have no message lookup on when a value is sent in a right inlet. However, these objects always do a message lookup on output. Therefore, it's always better, for integers, to use something like **select** or == if you're looking for a specific number. An example of this will be in a moment.

If you're not running in **Overdrive** mode, graphic objects slow you down because it takes time to redraw the screen. If you are in **Overdrive**, they don't slow you down, unless there's a message lookup involved. There is no message lookup with **Number boxes**, for example, because they handle only numbers.

If you want to filter MIDI messages according to channel, it's always better to use a channel argument in the MIDI receiving objects than it is to try and use the channel number output to route information later.

A message lookup is always performed on the output of message boxes. Therefore, it's better to type a number into an object box—which creates an **int** object—if you want to produce a constant value in an efficiency-conscious program. Of course you have to send bang to such an object

(whereas a message box can be triggered by a variety of messages in its inlet), but if this can be arranged, it's faster than using a message box. In the vast majority of cases, the difference in speed is negligible, but if enough instances like this are added up, they can have a noticeable effect.

As for keeping down the memory usage of a large loaded patch, certain objects store data in less efficient ways. For example, if you need to store three numbers together, it's not efficient to use a **table** object, even one set to a size of 3 elements, because each instance of a **table** stores a lot of additional information that uses up more memory. If you don't necessarily need all of the features of a certain object, you might try to use a simpler object that will use less memory.

Memory Usage

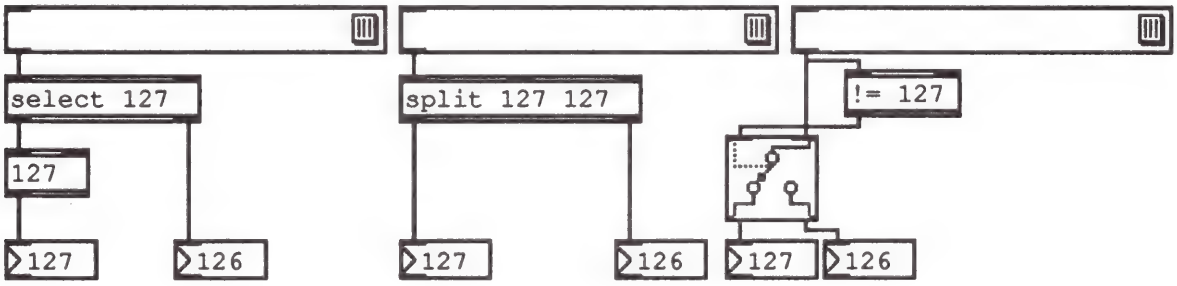
If you have written a rather large program (and especially if you have a Macintosh with only 1MB of memory) you will want to try and keep down the amount of memory your patch uses when it is loaded. Doing so will also make your patch load faster.

Try to avoid doing similar tasks with many copies of a single subpatch, since copies of all the objects contained within the subpatch are created for each instance of the subpatch you use. It is better to design your subpatch to work with a variety of incoming values than to use the #1-#9 argument feature to differentiate 50 copies of a subpatch.

There is a memory overhead of at least 100 bytes for every visible box on the screen, though boxes in closed windows take up less space.

Examples

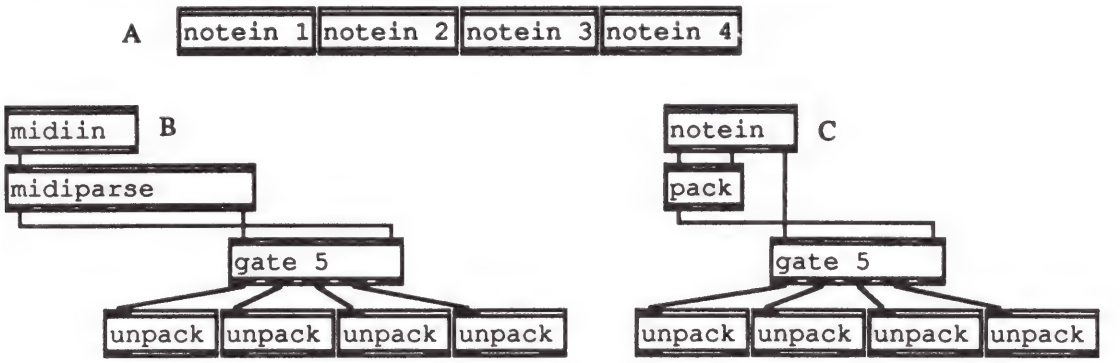
Three ways to send 127 to one place and everything else to another place. The method on the right is marginally slower since it involves the graphic object gGate, and message lookup (at the output of gGate).



Message lookup is a factor in computation speed

Efficiency

Three ways of getting (and separating) note data from 4 different MIDI channels



Method A is the most compact and efficient, both in memory and speed

See Also

Encapsulation How much should a patch do?

Encapsulation

How much should a patch do?

Complex Patches

Once you start writing relatively complicated programs, try to build them out of different parts, rather than one enormous, tangled patch in a single Patcher window. The way to do this is to divide your program up into different Patcher files. The different files can be subpatches of one *main* patch, so that they are all loaded when the main patch is opened.

Subpatches can communicate with each other via inlets and outlets, or **send** and **receive** objects, and they can share data by using **coll**, **table**, or **value** objects which have the same name as an argument. There is no reason that a large and complicated program cannot be composed of many smaller parts, and the advantages of this approach are considerable.

Modularity

There are several important reasons why it is a good idea to use a *modular* approach to programming. One reason is that it makes it easier to verify that your program actually works, especially in extreme or unusual cases. This becomes harder and harder to do as a program grows in size and complexity. Therefore, by building small modules and ensuring that each one works as its supposed to *in and of itself*, you reduce the number of possible problem spots when the modules are combined in a larger context.

A second reason is that many tasks in a program are used again in different contexts. Once you have built a small module that performs a certain task, you can use that module wherever the need for that task arises, rather than rewriting it each time.

Another reason is that many tasks in a program are *similar* to other tasks. By writing a small, general-purpose module (usually one that takes arguments so that its exact function can be modified by the argument), you can use that one module with different inputs or arguments, to do many similar things.

Finally, by encapsulating different portions of the program, you make it easier for yourself (or others) to see how the program works long after you develop it.

Encapsulation

The different modules of a program are best designed to *encapsulate* a single task. Name the module for what it does, and reuse the module should you ever with to perform the task again in another program.

By keeping certain values in one place, you only have to change them once if you decide they need to be modified. If the same values are distributed throughout your program, you have to find every instance of that value, and change each one individually.

One way to keep values in a single place, yet still make them available to many different objects is to store the values in a single file that can be accessed by any patch. For example, many different patches can read in values from the same **table** file by using table objects with the same filename as an argument. Changing the contents of *one* file changes the values used by *all* the patches, that share the files.

How much should a patch do?

Encapsulation

Messages between Modules

When designing small modules which will be combined in a larger program, it is important to consider not only what the module does internally, but also the context in which it will be used. The context will determine what kind of messages you want each module to produce and accept. For example, you might wish to use a bang to trigger a process, numbers to toggle something on and off or to provide values for calculation, or symbols (such as start and stop) to control a more complex task such as a sequencer.

The simpler the messages that a module receives and sends, and the simpler the function of each module, the greater the number of contexts in which that module is likely to be effective.

Documenting Subpatches

Here are three tips for documenting your own patches that will be used as subpatches:

1. Give your subpatches informative names, so you'll remember what each one does.

2. Put Assistance text in each **inlet** and **outlet** object, to remind you of the inlet or outlet's purpose when using the patch.

3. If your subpatch is complicated, include **comments** inside it to explain its operation.

See Also

Debugging	Techniques for debugging patches
Efficiency	Issues of programming style

Errors

Explanation of error messages

Error Reports in the Max Window

Max prints error report in the Max window when you make a programming mistake. Below is a list of error messages you may encounter, along with likely causes of each message.

"$" variable out of range
Occurs when you refer to an argument number out of the range $1-$9 in a message sent to a message box.

<filename>: can't open
Occurs when a patch is loading or when an object is created that reads its data from a separate file. The file that was to be read in automatically was not found in Max's search path or was not a type of file that the object is capable of opening. The erroneous filename has usually been specified as an argument to an object such as **coll**, **seq** or **table**. Make sure that the file is in Max's search path.

<objectname>: bad argument
Occurs when creating a new object with typed-in arguments. There is something wrong with what you typed after the name of an object. Usually the object is expecting a symbol, and you typed in a number, or vice versa. Check the object's argument specification list in the *Objects* section.

<objectname>: bad arguments for message <message>
Occurs when an object receives a message that it understands, but one of the arguments in the message is not what the receiving object expected. Usually the object was expecting a symbol argument and got a number, or vice versa. Check the object's input list in the *Objects* section.

<objectname>: doesn't understand <message selector>
Occurs when an object receives a message that it doesn't expect. It is possible to make patch cord connections that will result in improper messages being sent to an inlet. For example, Max will let you connect the outlet of a message box to almost any inlet, because there's no way of knowing what message will come out of the outlet. In such a case, the error does not become evident until you test the program and the message is actually sent.

<objectname>: message too long <message>
A message was sent that contained more than 256 elements.

<objectname>: missing arguments for message <message>
Occurs when an object receives a message that it understands, but one or more of the expected arguments in the message is missing. Check the object's input description in the *Objects* section.

Errors

<objectname>: No such object
Occurs when creating a new object or loading a document. When you are editing a Patcher, and you type the name of a nonexistent object into an object box (or the name of an object or subpatch that is not in Max's search path), Max produces this error message.

When you open a document that contains an object that Max cannot find (either because it is not located in Max's search path or because it just doesn't exist), Max displays the object as if it existed, except that the object name is surrounded by a dotted outline in the object box, and an error message is printed in the Max window. This preserves the connections to the object box in case you can retype the object to create it properly.

<symbol>: bad arg types
Occurs when a patch is running and a symbol is received in the inlet of a bitwise operator such as **&**, **|**, **<<**, or **>>**. Make sure that only number messages are sent to bitwise operators.

<symbol>: bad port argument
Occurs when an invalid port argument is typed into the object box of a MIDI object, or when a Patcher file is loaded that contains a MIDI object with an invalid port argument. Only the letters a-f are valid port arguments when the MIDI Manager is being used. When the MIDI Manager is not being used, only the letters **a** and **b** are valid port arguments.

<symbol>: illegal name for send | receive | value
Occurs when you type #1-#9 as an argument to a **send**, **receive**, or **value** object. #1-#9 must be followed by additional characters. See the *Arguments* topic for more information.

<symbol>: no such object
Occurs when a message is sent to a **send** object or to a message box which contains a semicolon followed by the name of a receiver, and there is no **receive** object with the name specified in the **send** object or message box.

<number>: not a symbol
Occurs when the element which follows a semicolon in a message box (specifying a receiver for the message) is not a symbol.

bad message
Same as <objectname>: doesn't understand <message selector>.

bad receiver
Same as <objectname>: doesn't understand <message selector>.

bag | float | int | pack | table: missing or incorrect arguments to send
Occurs when the patch is running and a **bag**, **float**, **int**, **pack**, or **table** object receives a send message without an argument, or with an argument that is not a symbol or the name of an existing **receive** object.

Errors

Explanation of error messages

can't connect <objectname> to <objectname>
Advisory message produced when you try to connect an outlet to an inlet that doesn't understand the message sent by the outlet. The first thing you should have noticed was that the inlet did not "light up" when you dragged the mouse over it.

consistency check failed:
<message>
Occurs when there is a bug in the Max application or in an external object. Please report the contents and context of any such message to Opcode.

cannot type <objectname> into an object box
Occurs when you try to type the name of a user interface object (an object that has its own graphic icon such as a **slider** or **button**) into an object box.

Error loading external file <filename>
Occurs when Max is installing an external object in the startup folder. The external object file is damaged. Try restoring a copy from the original disk.

funbuff: bad file type
funbuff: file not found
Occurs when a patch is loaded or when a **funbuff** object is created that reads in from a separate file. There was an error in reading a file into a **funbuff**, either because the file was not in the proper format (it must start with the word funbuff, followed by a space-separated list of numbers) or because a Max or text file with that name could not be found. Ensure that the file is in located in Max's search path, and that it is in the proper format.

grab: can only connect to leftmost inlet
Occurs when you try to connect the right outlet of a **grab** object to the wrong inlet of another object. The right outlet of **grab** should be connected only to the *leftmost* inlet of other objects.

graphic: <name> already exists
Occurs when you create a **graphic** object that has already been taken by another object, such as a **table** or **send/receive** pair.

inlet: wrong message or type
Occurs when a patch is running and an object receives a message that it doesn't expect in some inlet other than the left inlet.

lib: can't find script file <filename>
Occurs when a new **lib** object is created, or when a patch that contains a **lib** object is loaded. Max can't find a script file with the same name as was typed in as an argument to the **lib** object. The script file must be located in the same directory as the Patcher file which contains the **lib** object, or in any folder in Max's search path.

lib: error creating file
Occurs when the contents of a **lib** object are saved in a separate file (with a write message). The disk you are trying to save to is probably too full for Max to create the file. Create more free space on your hard disk.

Errors

lib: error evaluating script
Occurs when creating a new **lib** object, or when loading a Patcher that contains a **lib** object. You have made an error in the format of the script file that the **lib** object is using. Check your script file for errors, and consult the *Librarians* topic the correct format of a script file.

lib: load cannot occur at interrupt level
Occurs when **Overdrive** is enabled, and you send the load message to a **lib** object from a timing object such as **metro** or **pipe**.

lib: requires script file name argument
Occurs when creating a new **lib** object without typing in the name of a script file as an argument. The argument is obligatory.

patcher connect: inlet <number> out of range
Occurs when editing the name or arguments of an object that has already been created in a Patcher, and patch cords that used to be connected to the object can no longer be connected. Changing the contents of the object box may change an object's number of inlets or outlets, or Max may be unable to create the object at all if you type in the wrong thing.

read failed
Occurs when a file is read into an object. Max encountered an error reading a file and could not load in the data. Check to make sure that the file is in the proper format for the object reading it in.

rescopy: failed to add XXXX N, error N
Occurs when installing an external object. The disk that Max is running on is probably too full for Max to create the Max Temp file that it uses while the application is open. This is a serious problem and we suggest you quit Max and create more free space on your disk. Typically, you'll need at least 30K, although much of this space is used by the **led** object, so if you're extremely tight on disk space, don't use this object.

rescopy: failed to get <resource type> <ID number>
Occurs when installing an external object. The external object file is corrupted. Restore a new copy of the external object from your original disk.

route: argument types don't match
Occurs when you type in a mix of symbols and numbers as arguments to a single route object. The arguments must be either all numbers or all symbols.

route: too many arguments
Occurs when you type in more than 10 arguments to a **route** object. The rightmost outlet of both the **sel** and the **route** objects is an "overflow" outlet, which passes on any messages which are not matched. Multiple **sels** and **routes** can be linked, if you need more than 10 arguments.

send: <symbol>: already exists
Occurs when you type in a name as an argument to a **send** object which is already being used for a **table** or other object.

sxformat: illegal type in message
Occurs when the patch is running and some message other than an int is received in the inlet of **sxformat**.

textbox: bad args
Occurs when opening a Max document. The document has been damaged.

warning: extra arguments for message
Occurs when an object is given more typed-in arguments than it expects, or when too many arguments are present in an incoming message. Usually this is just a warning of something that's not quite right but is basically harmless.

Error Dialogs

When an error occurs that requires your immediate attention, the error is reported in a dialog box. The following errors can appear in dialogs.

Choose Resume from the Max menu to restart the Max scheduler...
It is possible to get Max working so hard it doesn't have time to respond to your commands (say, if you have a number of **metro** objects sending out bangs as fast as they can, or if you have created a loop which overloads Max, causing a Stack Overflow error). Holding down the command key and typing a period (⌘ - .) will stop Max's scheduler, giving you time to turn off some of the overloading processes. When Max's timer is stopped, the above message is shown in a dialog box. Choose **Resume** from the Max menu to restart Max's timer.

Error creating <filename>
Occurs when Max is unable to save a file as you had requested. Check to see if the volume you are trying to save to is write-protected.

Error writing <filename>
Occurs when Max is unable to save a file as you had requested, probably because the disk is too full. Create some free space on your disk, or save to a different disk.

Error opening filename
Occurs when Max is unable to open a text file as you had requested, because Max can't find the file or can't read it properly. Make sure that the file is a Max document or a standard text file.

Stack Overflow
Occurs when an object's output is being fed back into its inlet in some type of loop. After stopping the process that is causing the stack overflow, choose **Resume** from the Max menu to restart Max's scheduler.

Explanation of error messages

Errors

See Also

Debugging	Techniques for debugging patches

Externals

What are external objects?

Two Kinds of Objects

In various places in this manual, you'll see references to *built-in* and *external* objects. The differences are minor but important. Built-in objects are part of the Max application. Each object is a (relatively) small amount of C code, but since there are about 100 built-in objects, one can see why the size of the Max application file is around 500K.

External objects are separate files which contain C code that can be "linked into" the Max environment in such a way that it is impossible to tell that the code was not part of Max in the first place. This is why we rarely mention whether any particular object is a built-in or an external in this manual. External objects are something like Hypercard XCMDs, but unlike XCMDs, which are structured as "procedures" Max external objects have the status of "real objects" that receive messages just like those built into the application.

There are two kinds of external objects, *normal* objects whose names you type into an Object Box, and external *user interface* objects whose icons appear in the Patcher window palette and have some distinctive graphical appearance in a Patcher. External user interface objects should be placed in your startup folder so their icons are available in the palette from the time you begin working. Normal external objects can be placed in another folder in Max's search path and will loaded as they are needed.

Normal external objects can provide access to non-MIDI hardware devices, such as the **cd** object which controls the Apple CD-ROM drive. Or they implement capabilities not found in the basic Max set of objects. A good example is **coll**, which allows the storage and retreival of arbitrary list structures indexed by numbers or symbols. External user interface objects, such as the Keyboard Slider **kslider**, are important for constructing custom interfaces within Patcher windows.

Keep External Objects Where They Can Be Found

One point you should keep in mind about external object files is that Max needs to be able to find them when loading a document. This is especially important with external user interface objects. If you load a patch which contains references to missing external user interface objects, the wiring of the patch may get quite screwed up when the patch is opened. If the patch contains embedded presets, missing user interface objects could result in a system crash, or at the very least a large stream of unhappy error messages in the Max window.

External objects can be manually loaded into the Max environment by choosing **Install...** from the File menu. Use the standard open file dialog to locate the external object file you wish to install. Note that if you are developing an external object, you cannot use **Install...** to add a "revised" copy of an external object into the environment. Max will continue to use the first object definition until you restart the application.

If you expect an external object to be loaded automatically only when it is *referenced* (by loading a Max document that uses it, or typing its name into an object box), the name of the external object file must be consistent with the name of the object, even though the name of the object is derived from the name of a resource inside the file.

Overview of graphics windows and objects

Graphics

Introduction

Max contains a set of objects for color graphics and animation. These objects use the same principles as objects which are used for music processing, so a single Patcher can combine user interface, music, and graphics functions. This allows you to experiment with various ways of combining and synchronizing sound and image.

Graphics objects draw within special *Graphics Windows*, associated with a **graphic** object. Each **graphic** has a name, and each object which draws something must have the name of a graphics window as an argument.

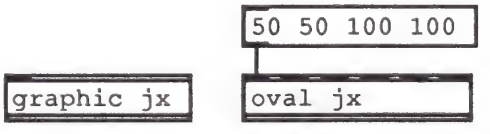
When Max is in **Overdrive** mode, objects that draw in graphics windows are de-prioritized, so that a process which plays both music and displays graphics can run at any speed and the music will not be affected by the speed of the display. For example, if an animation would ordinarily "run behind" the music, the animation automatically skips frames to keep up. User interface objects such as sliders do this too, by the way.

All the objects which draw in graphics windows are external objects, and additional graphics objects can be written by C programmers. Each object is a *sprite* associated with a particular window. Sprites allow objects to pass in front or behind each other according to a *priority number*. Higher-numbered sprites are drawn in front of lower numbered sprites. The priority number of most of the graphics objects can be changed with the priority message.

Making Graphics Happen

Once you have a patch containing a **graphic** object, you need one or more drawing objects. There are three basic drawing objects included with Max: members of the **oval** family (**oval**, **rect**, **ring**, and **frame**) which draw shapes; **pics**,which displays PICS animation files; and **pict**, which displays PICT files. The first argument of any drawing object is the name of the **graphic** object whose window will be used for drawing. The **graphic** object need not exist at the time the drawing object is created, but the drawing object will do nothing until there is a valid (and visible) graphics window with the same name specified as drawing object's argument.

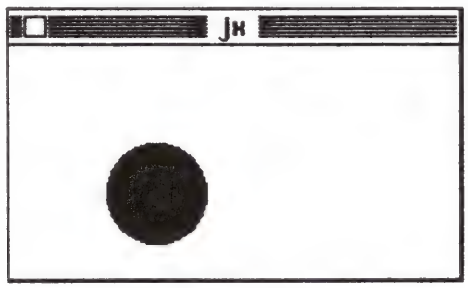
Here is a simple patch which draws a black oval at 50,50,100,100 in the graphics window jx when the user clicks on the button. The oval has six inlets, for left, top, right, and bottom screen location, drawing mode, and color (index into the graphics window's palette). A list of four numbers sent to an oval causes it to draw at the coordinates specified in the list: left, top, right, and bottom.



Graphics

Overview of graphics windows and objects

A graphics window titled *jx* will appear when the **graphic** object is created. Below is shown the result of clicking on the message box in this program.

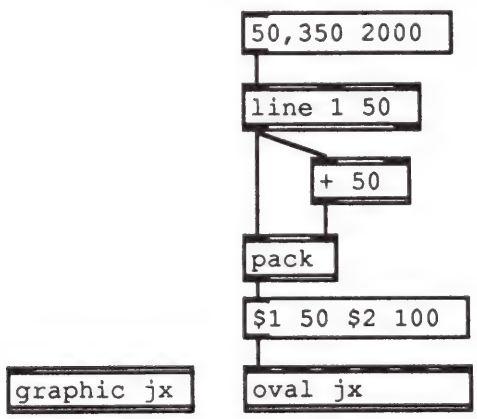


If the **oval** is redrawn with different coodinates, the old oval is erased automatically, because the oval acts as a sprite which has changed location (and possibly size).

Ways to Move Objects

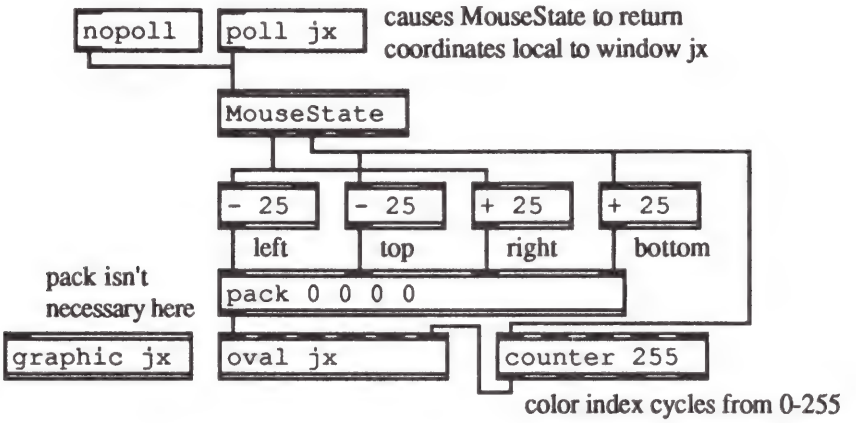
Two useful objects that let you move a sprite-based object such as an **oval** across the screen are **line** and **MouseState**. **line** can move the sprite smoothly in a trajectory, while **MouseState** can be used to make a sprite follow the mouse.

Here is a program which uses **line** to move an **oval** from one side of the window to the other. Notice that only the left and right coordinates are changed by the output of the **line** object.



Graphics

The following program uses **MouseState** to follow the mouse. When **MouseState** receives the poll message with the name of a **graphic** object has an argument, it will begin polling the mouse when the associated Graphics Window becomes the active window (or, if **All Windows Active** is enabled, it polls all the time). The local coordinates of the mouse in the graphics window are sent out the second (horizontal) and third (vertical) outlets of **MouseState**.This program draws an oval centered at the mouse location whose color changes each time the mouse is moved.



See Also

graphic	Graphics window object
oval	Draw shape in graphics window
MouseState	Report the status and location of the mouse
pics	Animation in graphics window
pict	Draw picture in graphics window

Librarians

Creating patch editors and librarians

Introduction

The **lib** object is a script-driven generic librarian. It doesn't handle multiple patch types or references between patch types (such as a "configuration" which would need to know the "program" names). However, it does make it somewhat easier to write *simple* editors and librarians. Creating an editor/librarian in Max is done in three parts:

First, you write a script (as a text file) which is read in by a **lib** object when it's created.

Second, you situate the **lib** object (with the name of the script file as an argument) in a Max patch, following the general guidelines shown in the example in description of **lib** in the *Objects* section.

Third, you create an *editor* — a Patcher which uses special objects **libto**, **libfrom**, and **libswitch** to communicate with a librarian and provide a user interface to a particular patch's parameters.

Format of a Script File

The **lib** object's script file specification isn't terribly user-friendly, but there's no reason why a dialog-like interface couldn't be created to make generating the files a bit easier. It hasn't been done yet, however, so you'll have to learn the **lib** script "language."

Each line in the lib script starts with `#L`. Don't ask why, but it does.

Following the `#L` is a keyword, such as outbank, which will define a particular aspect of the librarian's behavior. Additional information for each keyword may follow after that. Each line must end with a semicolon. If a "line" is too long for a single line of text, you may put it on two lines. But always end with a semicolon.

```
#L keyword <additional stuff>;
```

The first keyword in every script file should be define and the last end.

define *type-id window-name*

Starts the definition of a librarian. *type-id* is a four character identifier which will be used to distinguish windows and files for this device. *window-name* is the text which will appear at the top of a library window. For example...

```
#L define m3rP M3R Programs;
```

...will produce a library window like this:

The files saved by this librarian will have the file type `m3rP`.

end

Ends the definition of a librarian.

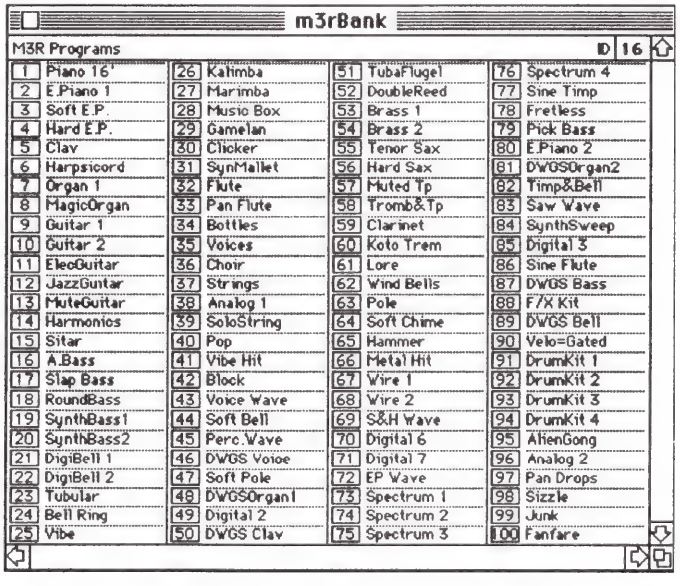
patchnumbers *starting-number number-of-patches numbering-system*

Specifies how patches are numbered, starting at *starting-number* (either 0 or 1). *number-of-patches* is the total number of patches in the group that the lib will be receiving. *numbering-system* is 1 for decimal, 2 for 11-88 "LED" format, and 3 for hexadecimal. For best results with numbering system 2, the starting patch number should be 0.

rows *number-of-rows*

columns *number-of-columns*

Specifies the number of displayed rows and columns in the library window. *number-of-rows* * *number-of-columns* should be equal to the number of patches above. For example, a window with 25 rows and 4 columns (100 total patches) would look like this:



compress *method*

Specifies how the data is transformed from a series of 7-bit numbers in the system exclusive dump message into its internal format in memory (and vice versa). Often the compressed size will be a different number of bytes (either larger or smaller). "Internal" compression methods are specified by number. Other compression methods can be specified by name (for example `#L compress KorgMeth`) and written as C code resources. If no compression is needed, you can skip the compress statement.

Internal Method Numbers:

1 7-into-8, 8-into-7, where the high bits of the subsequent 7 numbers are placed in the first 7-bit number (of 8).

2 8 bits sent as two 4-bit nibbles, lowest 4 bits first

3 8 bits sent as two 4-bit nibbles, highest 4 bits first

4 Not implemented

5 Converts DX7 128-byte format into 155-byte format

For C programmers, remember that you can invent any format you want to compress the data by writing a code resource. An example code resource, which performs the same compression as internal method number 2, is included in the development materials supplied with Max.

Note that many synthesizers don't need compression methods. Most Roland equipment is designed this way.

sizebank *size-of-bank-in-bytes*

Specifies the size of a compressed bank in bytes.

sizepatch *size-of-patch-in-bytes*

Specifies the size of a compressed patch in bytes.

patchnames *start-index end-index*

Specifies the starting and ending index of the patch name that is included within a patch. For example, if the patch name consisted of the first 10 characters, you would use the statement...

```
#L patchnames 0 9;
```

Note that these indices are in the compressed patch, not the data bytes of a patch as they are received. If patch names are not included within the patch, use the excluded statement instead of patchnames.

excluded *length-of-name*

Specifies that the patch type doesn't contain names, and you would like names "tagged" to each patch when stored in files and displayed on-screen. *length-of-name* specifies the number of bytes to be given to each name. A length between 10 and 18 characters is reasonable.

columnwidth *column-width-in-pixels*

Specifies the width of a column in the library window. 80 pixels is a reasonable width (and is used as a default if you don't include a columnwidth statement in your script).

sendonselect

Specifies that the currently selected patch in a library window should be sent to the synthesizer.

aspatches

Specifies that the MIDI device does not have any "bank" level system exclusive dump messages. This modifies the meaning of the MIDI Communication statements discussed below. Examples of devices which operate this way are the CZ-101 and the PCM70.

editor *patcher-name*

Defines patcher *patcher-name* as a patcher which will be loaded when the user double-clicks on the number of a patch in the librarian. The editor Patcher file should be located in the same folder as the file which contains the **lib** object or in the Max search path. Refer to the descriptions of **libto**, **libfrom**, and **libswitch** for information about how to create an editor.

MIDI Communication Statements

Each of the MIDI Communication Statements specifies a particular protocol for sending and receiving data. For sending, the statements define what bytes to send. For receiving, the statements define a pattern that the **lib** object will look for to capture the data in an incoming system exclusive message. Since many **lib** objects might all be listening to the same MIDI input, it's important to be able to distinguish one device's sysex from another. This can even be done with multiple devices connected to the same input, since lib objects can distinguish the MIDI channel (or "device ID") number if necessary.

Communication Statements are made up of a keyword followed by "sub-statements" which are separated by a slash (/) surrounded by spaces. An entire statement might look like this:

```
#L outbank 0xF0 0x42 / chan 0x30 / 0x24 0x4C / data 8572 / dc / 0xF7;
```

Definitions of Communication Sub-Statements

(Numbers)

If sending or expecting literal numbers (such as the start or end of system exclusive), a string of numbers constitute a sub-statement. Hex numbers are preceded with a 0x. Example:

```
0xF0 0x42 0x24 45 0xF7
```

dc

Expect a "don't care" byte. Only useful for input, where the information is not needed and you don't want to match it. For example, often the size of the data in the bank or patch is sent immediately before the data itself. When the size is always constant (and you already know what it will be) this is irrelevant data which can be matched with a dc statement. Each dc statement matches a single MIDI byte.

chan *mask-value*

Send or expect to match the MIDI channel (as set with the ID numerical in the library window, and/or specified as an argument to the lib object). *mask-value* is a value which is or-ed into the channel value (which will range from 0 to 15 for channels 1 to 16). For example, you might see something in a device's MIDI specification that looked something like...

3cH (where c = device MIDI channel)

This usually means that the high nibble is always 3, and the lower nibble contains the MIDI channel. This could be sent or matched with:

```
chan 0x30
```

checksum *method-number*

Send a checksum of the data bytes (for incoming data, this byte will be ignored). *method-number* is either 1 for a straight sum of the data bytes (modulo 128), or 2 for a 2's compliment of the sum.

data *bytes-to-send-or-expect*

Send or capture the data bytes in a bank or patch. The size should be the uncompressed number of bytes in the patch or bank. When receiving, the bytes will be captured and subsequently compressed (if applicable). When sending, the bytes are uncompressed (if applicable) and sent at this point in the communication statement.

is *e xpr-statement*

The is keyword can be followed by a C-like expression which is evaluated by the Max **expr** object. Several of the "argument" variables (refer to the description of **expr**) have been defined for use in **lib** script files. The first four only apply to parameter change messages. They are:

$i1 The "parameter number" as defined by the param argument to **libto** (explained in the description of the **libto** object)

$i2 The parameter value from **libto**

$i3 Not used

$i4 The index into the compressed patch buffer of the parameter as defined by the argument to **libto**

$i5 The current MIDI channel

$i6 The current patch number being sent or received

An example of an is-expression is found under the definition of the param message below.

When **lib** is receiving data, the expression will be evaluated and if the incoming byte matches the result of the statement, the pointer will move to the next byte in the matching process. There is a limitation of 32 is statements in a single **lib** script file. If you include a slash in an is expression statement, be sure to put a backslash before it.

mode *mode-statement-number*

mode allows you to define a script "macro" which can be inserted into a transmission statement. mode is designed for synthesizers that need to be put into a special mode with a MIDI message before they're ready to receive system exclusive data. You can have up to 8 mode statements, numbered 1 to 8. Here's an example:

```
#L mode 1 0xF0 0x42 / chan 0x30 / 0x24 0x4E 0x04 0x10 0xF7;
```

The above statement *defines* mode number 1. The following statement will *use* mode number 1. It precedes the actual sending of a patch, insuring that the device is set to the mode in which it can receive the patch.

```
#L outpatch mode 1 / 0xF0 0x42 / chan 0x30 / 0x24 / data 86 / 0xF7;
```

Modes are currently not used in reception statements.

patchnum

When sending, patchnum is set to the number of the patch currently being sent. If the first patch number starts at 1 (as defined in the patchnumbers statement), this value will have had 1 subtracted from it.

MIDI Communication Keywords

In the words defined below, *message* refers to a statement made up sub-statements as described in the preceding section.

reqbank *message*

Specifies the data that will be sent to a device to request a bank. This data will be sent when the **lib** object receives the load message with no arguments.

reqpatch *message*

Specifies the data that will be sent to a device to request a patch. The patch number (for purposes of the patchnum sub-statement) is the argument when the **lib** object receives the load message, i.e. load 5 will request patch number 5. If it's only possible to load the "active" patch (as in a DX7), the presence of the argument to load serves to distinguish loading a patch from a bank.

inbank *message*

Specifies the data to be expected when a bank is being received. Each element of the message is matched in sequence with incoming MIDI data. At the point the messages differ, the matching is abandoned and the **lib** object waits to see the first byte of the inbank message in subsequent incoming data. If the entire message is seen, the data is placed in the **lib** object's buffer and displayed in its library window.

inpatch *message*

Specifies the data to be expected when a patch is being received.

outbank *message*

Specifies the data that will be sent to dump a bank to the device.

outpatch *message*

Specifies the data that will be sent to dump a patch to the device. This message is executed in response to "send on select" so the patch should be sent to the edit buffer of the device if possible.

param *message*

Specifies the data for a parameter change message. Parameter change messages are sent in response to numbers coming into **libto** objects or the send message sent to a **lib** object. You'll probably want to use is statements in defining a param message. Refer to *Definition of Communication Sub-Statements* above for information about is statements and the variables you can use inside them to refer to parameter numbers and values. Here is an example of a param message which sends system exclusive to change the Lexicon LXP-5. Note the way the parameter number ($i1) and parameter value ($i2) are incorporated into the message with an is statement and argument variables.

```
#L param 0xF0 6 5 / chan 0x20 / is $i1 / is $i2 & 0x7F / 0xF7;
```

mode *mode-number message*

Defines a mode message macro which can be referred to with the mode sub-statement. Up to eight modes, numbered 1 to 8, can be defined. Typically a mode message definition will be an entire

MIDI system exclusive message which puts a synthesizer in a particular mode. The **lib** object will not send a mode message if it was the most recent mode sent.

Considerations for aspatches Mode

aspatches mode loads or dumps an entire bank as a set of individual patch requests or dumps. In this mode, the outbank and inbank statements are designed to get a single patch. These statements use the patchnum sub-statement to specify which patch is being loaded or received. The **lib** object will send each message once for each patch in a bank. As an example, consider the PCM70, which only sends and receives one patch at a time. This is the outbank statement used in the PCM70 script.

```
#L outbank 0xF0 0x06 0 0x19 / patchnum / 0x02 0x4E /
          data 334 / checksum 1 / 0xF7;
```

When the **lib** object receives the dump message, the value of patchnum will range from 0 to 49 (there are 50 patch locations in the PCM70). In addition, **lib** will insert a short delay (100ms) between each message.

Overview of Writing Editors

The **lib** object has three "friends" which make it possible to connect a patcher to communicate with a library. Using the editor keyword in the script file, you can define a Max patcher file as an editor. When you double-click on the number of a patch, the editor window is loaded and opened. The patcher must be located in the same folder as the patcher which contains the parent **lib** object, or in another folder in Max's search path.

You use **libfrom** objects to get values from a patch in the library when the editor window opens and **libto** objects to put values back in the patch and (optionally) send parameter change messages. The **libswitch** object can be used to switch from one patch to another inside an editor window.

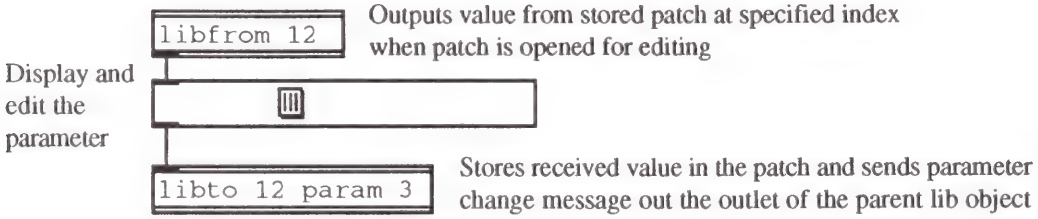
When creating an editor, we suggest the following procedure:

- Get your **lib** loading and dumping patches properly, and save a file with a bank of patches in it.
- Save an empty patcher with a name like myDeviceEdit.
- Add the param keyword followed by a parameter change message to your script, if the device accepts parameter changes. If the parameter is specified by its index into a patch, use $i4 to refer to the parameter "number" otherwise use $i1 and remember to use param and the proper parameter number in the arguments to each **libto** object you create.
- Add a statement with the editor keyword followed by the name of the saved empty patcher to your script
- Open or load a bank of patches, and double-click on one to open the editor patcher
- Unlock the window and begin creating the editor. The **libto**, **libswitch**, and **libfrom** objects you create will actually work in the proper context. If you created the patcher in isolation from its

librarian, these objects will be disabled, and you won't be able to test your work as go, which would be one of the benefits of using Max to develop an editor in the first place.

Note that **libfrom** and **libto** objects can be placed in subpatchers of the patcher named in the editor statement in the script file, but **libswitch** objects (which change the patch being edited) will only work if they are located in the top level Patcher window of your editor.

For each parameter you want to edit, you will need a configuration like the three objects shown below:


libfrom 12
Outputs value from stored patch at specified index when patch is opened for editing
Display and edit the parameter
libto 12 param 3
Stores received value in the patch and sends parameter change message out the outlet of the parent lib object

Example: An Entire Script File

This is a script for a librarian that deals with Programs for the Lexicon PCM70, which requires the aspatches style of loading and sending banks of patches. For best results, aspatches should be defined above any of the MIDI Communication statements.

```
#L define pcm70 PCM 70 Programs;
#L patchnumbers 0 50 1;
#L rows 25;
#L columns 2;
#L patchnames 3 15;
#L aspatches;
#L reqbank 0xF0 0x06 0 / chan 0x30 / 0x61 / patchnum / 0xF7;
#L reqpatch 0xF0 0x06 0 / chan 0x30 / 0x60 50 0xF7;
#L inbank 0xF0 0x06 0 / chan 0x10 / patchnum / 0x02 0x4E / data 334 /
   checksum 1 / 0xF7;
#L inpatch 0xF0 0x06 0 / chan 0 / 0x50 0x02 0x4E / data 334 /
   checksum 1 / 0xF7;
#L outbank 0xF0 0x06 0 / chan 0x10 / patchnum / 0x02 0x4E / data 334 /
   checksum 1 / 0xF7;
#L outpatch 0xF0 0x06 0 / chan 0 / 0x50 0x02 0x4E / data 334 /
   checksum 1 / 0xF7;
#L param 0xF0 / is $i2 / is $i4 / 0xF7;
#L sendonselect;
#L columnwidth 80;
#L sizepatch 167;
#L sizebank 8350;
#L editor pcmedit;
#L compress 2;
#L end;
```

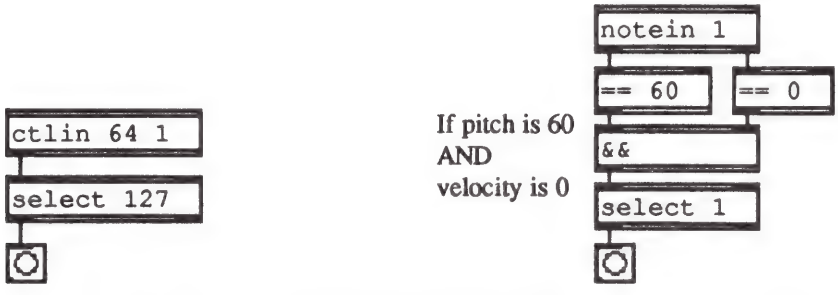
Ways to perform repeated operations

Loops

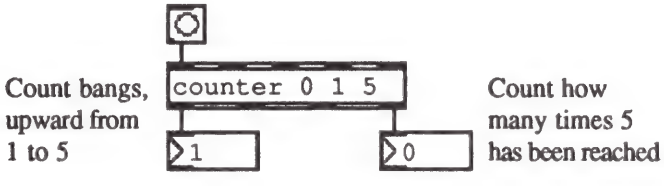
Repeated Actions

Many aspects of music-making involve repeated actions. For example, we might count 25 measures of rests, hit a gong 4 times in succession, repeat the whole section a total of 3 times, etc. In programming, repeated actions are called *loops*, because conceptually we think of the computer completing an action, then looping back to the place in its stored program where it started and performing the action again. A loop generally involves some sort of a check before or after each repetition, to see whether the action should be performed again (without the check, the process would continue endlessly).

In Max, a bang message can be used to signal that an event has occurred. In the example below, a bang is sent each time the sustain pedal (controller number 64) is pressed down, or each time the note Middle C (key number 60) is released.



Since bang is the generic indicator that something has happened, there is an object designed to count bang messages, called **counter**. It counts the bangs it receives in its inlet, and sends the count out its outlet. You can set minimum and maximum output values for **counter**, and set it to count up, down, up and down. In the example below, **counter** counts from 1 up to 5, then starts again at 1. The right outlet reports how many times the maximum (5) has been reached.

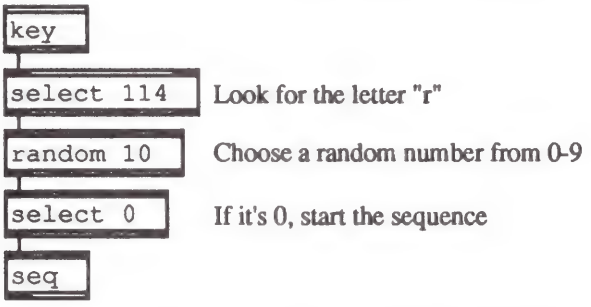


It can be useful just to send out a succession of numbers from a **counter**, for example, the numbers could be used as addresses to get values from a **table**. Other times, in order to make the loop useful, there needs to be a unique event when a certain condition is met. Actually, **counter** has its own built-in conditions and reactions, such as "when the maximum is reached, send the number of times it has been reached out the right outlet, send the number 1 out the right-middle outlet, and go back to the minimum," but sometimes we may want another condition to cause a certain result.

Loops

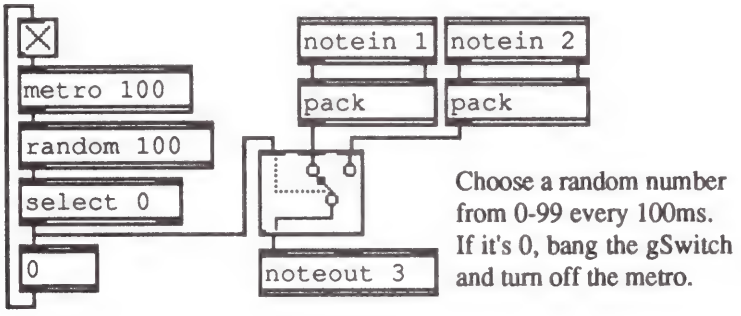
Ways to perform repeated operations

In the example below, a bang is sent to **random** every time the letter r is typed on the Macintosh keyboard. When **random** produces the number 0, a sequence is played.



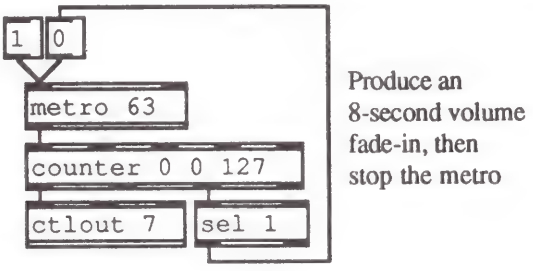
In this case, we aren't counting the number of times something happens (we might have to type the letter r any number of times before a 0 is chosen at random), we're just repeating *until* a certain condition is met.

When the condition we are testing for is met, something should happen as a result—a **gate** opened, a process started, a note played, etc.—and, if the repetitions are being supplied by a timed process (such as a **metro** sending a bang every 100 milliseconds), the repetitions should be stopped.



Timed Repetition

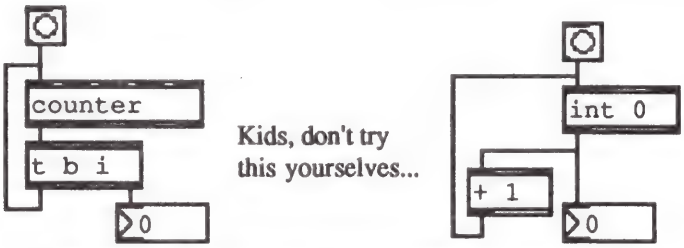
Since time is such an important factor in music, you'll want to have repeated actions occur at a specific speed. The **metro**, **clocker**, and **tempo** objects are used for producing output at regular intervals—bang in the case of **metro**, numbers in the case of **clocker** and **tempo**. (Of course, **metro** can also produce a succession of numbers when its output is sent to a **counter**.)



Loops

Stack Overflow

Automatic timed repetitions must be separated by at least a millisecond or two, otherwise Max will generate a *Stack Overflow* , which stops Max's internal scheduler until you shut off the repetitions. Below are a couple of examples of what *not* to do, because you will cause a stack overflow.



These may look like good ways to send out numbers as fast as possible, but they will result in stack overflow.

See Also

clocker	Report the elapsed time, at regular intervals
counter	Count the bangs received, output the count
metro	Output bang, at regular intervals
tempo	Output numbers at a metronomic tempo

File Menu

New Patcher Create a new Patcher window. Same as the **Patcher** command in the New menu.

Open... Open an existing Max document. Max automatically determines the proper type of window in which to display the document—Patcher, Table, or Text.

Open As Text... Force Max to open an existing document in a Text window for editing. Any Max document or plain text file can be opened with this command.

Close Close the active (foreground) window. If changes have been made to the document since it was last saved, Max will ask you if you want to save those changes. If changes have been made to a *named* **table** contained in the document, and the *Don't Save* option for that **table** is not checked, Max will also ask if you want to save the changes made to the **table**. Changes to a named **table** are saved in the separate file bearing that name.

Save Save the active document. The values stored in **Number box**es and other storage locations are not saved with a Patcher document. Nor does Max save the values that are stored in unnamed **table**s within a Patcher, unless the *Save With Patcher* option is checked for that **table**.

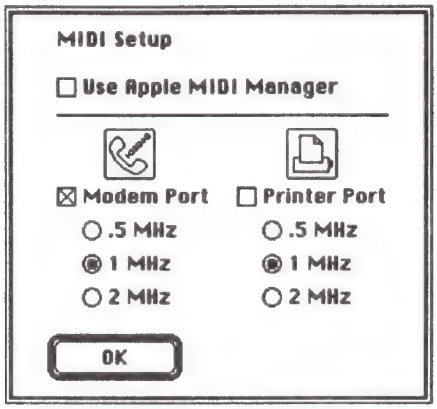
When you save an Untitled document, Max automatically uses the **Save As...** command, so that you can rename the document.

Save As... Save the active document in a separate file with a different name. The document is saved under the new name, and the document with the old name is closed unchanged.

The **Save As...** dialog also has a *Save As Text* option. Files that are saved as text can be opened and edited by any other application that reads text files.

Midi Setup... Choose serial ports and MIDI interface speed. If you wish to use the MIDI Manager, check the box next to *Use Apple MIDI Manager*. When using the MIDI Manager, connections to the serial ports are specified in the PatchBay application.

If you are not using the MIDI Manager, check *Modem Port* and/or *Printer Port*, whichever you wish to use for MIDI, and select the appropriate interface speed. (*1MHz* is the most common MIDI interface speed.)



The **Midi Setup...** dialog box is presented to you the first time you use Max. From then on, Max remembers the specified setup in the Max Preferences file.

Page Setup... Specify paper size, paper orientation, and printing options, before issuing a **Print...** command. If the patch you wish to print exceeds the bounds of the paper, you can reduce the image by specifying a percentage of reduction.

Print... Print the contents of the active Patcher or Text windows.

Install... Install Max external objects. When the Max application is opened, it installs external objects in its *Startup Directory*. Other external objects can be installed later with using **Install...**. If an external object is in Max's search path, you can just type its name into an object box to install it.

Quit Quit Max. If you have any unsaved windows, you'll be asked if you wish to save changes to the windows.

Edit Menu

Undo Undo the most recent editing change. Editing changes which can be undone are **Undo**, **Cut**, **Copy**, **Clear**, **Fix Width**, **Align**, changing an object's font and size, and moving an object.

Cut Delete the selection and put it in the Clipboard.

Copy Copy the selection to the Clipboard without deleting it.

Paste Paste the contents of the Clipboard into the active window. When you move objects from one window to another, Max will paste the objects into the new window in the same position they had in the old window.

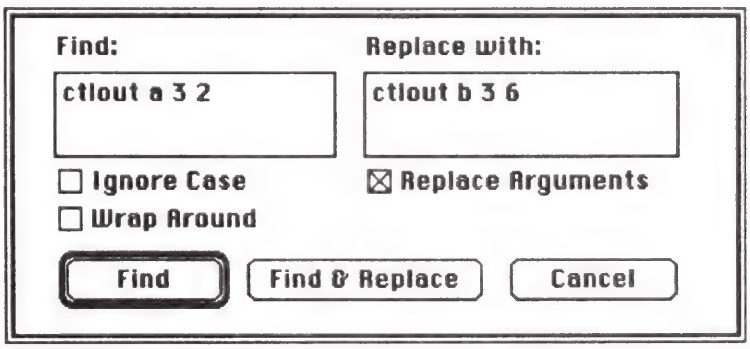
Clear Delete the selection without putting it in the Clipboard. Pressing the Delete (Backspace) key has the same effect in a Patcher window.

Duplicate — Duplicate the selection. The duplicated objects are placed just below and to the right of the original objects, and are automatically highlighted so that they can be dragged to the desired location. After the duplicate objects have been dragged, Max takes note of their position relative to the original objects. If you then *re*-duplicate the objects, without deselecting them, Max will put the new duplicates in the same position relative to the selection.

Select All — Select all objects in the active window. In a table editing window selects all the values of the table.

Paste Picture — Import a graphic image into a Patcher window from another application. Graphics can be made functional by placing the transparent button object, **ubutton**, over the image.

Find... — Find a string of characters in the active window. **Find...** search for a certain word or words and optionally replaces them with other words. This is especially useful in either Text windows or Patcher windows that contain many instances of the same word or words.



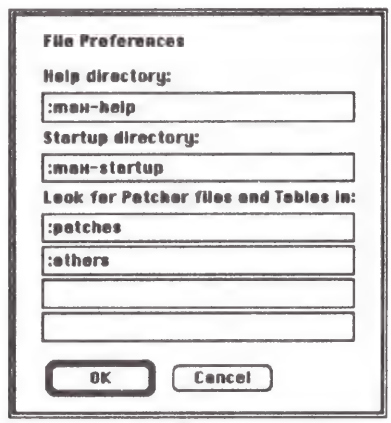
When the *Ignore Case* option is checked, Max pays no attention to whether letters are upper-case or lower-case. Max will normally beep when it has found all instances of the search string. If the *Wrap Around* option is checked, however, Max will go back to the first instance of the string after it has found all instances. The *Wrap Around* option works only in Patcher windows, not in Text windows.

The *Replace Arguments* option is for finding and replacing the argument of objects in a Patcher window. When this option is checked, and the search string begins with the name of an object, the replace string will replace the entire contents of that object. When Replace Arguments is not checked, the replace string will replace only the number of arguments in the replace string itself, leaving the rest unchanged. Depending on which button is clicked in the dialog box, Max can either *Find* the search string, selecting the search string in the active window when it is found, or it can *Find & Replace* the string, automatically replacing the found string with the replace string.

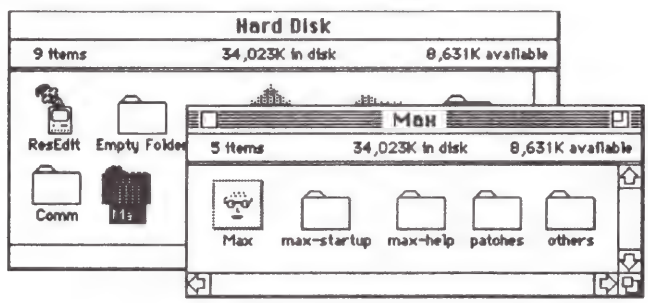
Find Again — Repeat the most recent search performed with **Find...**, finding the next instance of the search string.

Replace Replace the selected text with the replace string from the **Find...** dialog box, then repeats the search. Using **Replace**, you can step through each instance of a string, replacing only those you want to replace.

File Preferences... Opens a dialog where you set the directories (folders) that will be searched when Max looks for documents. The *Help Directory* is where on-line help files for each object are stored. Max loads all external objects contained in the *Startup Directory* when the program starts up. And the four directories you can list under *Look for Patcher files and Tables In:* are places to store external objects or other types of Max document you want to be able to access by name inside a Patcher program.



If a colon precedes the directory name, it is a *subdirectory* of the directory in which the Max application resides. The setup in the dialog box above corresponds to the arrangement of folders in the Finder shown below.



Use additional colons to specify directories within directories. For example, a subdirectory closet of the patches directory shown above would be specified as :patches:closet. If a name does not begin with a colon, it must be the name of a volume (disk or disk partition), and any directories in the volume are separated by colons. For example, if you want Max to look in a folder called Heap on the disk named Dump, specify it as Heap:Dump.

Menus

Explanation of commands

New Menu

Patcher — Create a new Patcher window.

Table — Create a new Table window, to make a **table** by drawing in values.

Text — Create a new Text window, to enter values in a text file, or simply to use as a note pad. Often values can be entered most efficiently by typing them in as text. For example, a **table** file can be created just by typing the word table, followed by a space-separated list of values. Once such a text file has been saved, it can be read by **table** objects in a patch. A **funbuff** file can be created by typing the word funbuff, followed by a space-separated list of alternating x and y values.

Max Menu

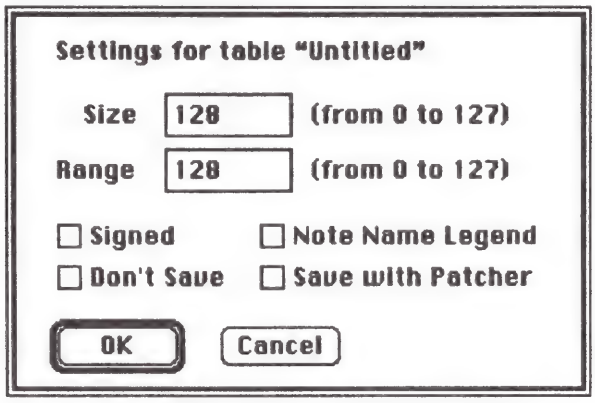
Edit — When **Edit** is checked, the active Patcher window can be edited. When **Edit** is unchecked, the active Patcher window is *locked*, and its user interface can be operated. Locking and unlocking the active window by clicking on the Lock button in the window's title bar has the same effect as choosing **Edit** from the Max menu. Yet another way to lock and unlock the active window is to hold down the command key while clicking in white space within the window.

Fix Width — Adjust the width of an object box, a **comment**, or a message box to accommodate all the text it contains on a single line. When **Auto Fix Width** is checked in the Options menu, Max will automatically adjust the width of object and message boxes each time they are edited.

Align — Adjust the position of all selected objects so that they are perfectly aligned vertically or horizontally. All selected objects are aligned in relation to the leftmost object in the selection.

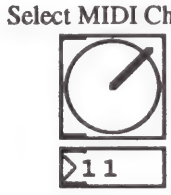
Max makes an educated guess as to whether to align the selected objects vertically or horizontally. If all objects in the selection are below the top edge of the leftmost object, but not completely to the right of the right edge of the leftmost object, Max aligns the objects vertically. Otherwise, Max aligns the objects horizontally.

Get Info... — Open a dialog box which lets you change special options for an object. Almost all of the user interface objects, as well as **tables** and some other objects, have certain characteristics (such as size and range) which can be set with **Get Info...**. The actual contents of the **Get Info...** dialog box depend on the object.

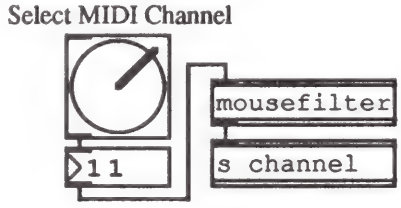


Get Info... dialog for a **table**

Hide On Lock — Set the selected objects and patch cords to be invisible when the Patcher window is locked, which reduces the cluttered appearance in a patch. Note that user interface objects which are hidden will not respond to mouse clicks.



Some objects and cords hidden



Nothing hidden

Show On Lock — Set the selected objects and patch cords to be visible when the Patcher window is locked, undoing the action of **Hide On Lock**.

Help... — Choosing **Help...** when an object is selected opens a help file describing that object. To get help about all objects, choose **Help...** with no object selected. You can get help on any object at any time by option-clicking on its box.

Resume — It is possible to get Max working so hard and fast that it doesn't have time to respond to your commands if you have a number of **metro** objects sending out messages as fast as they can, or if you have created a loop causes a Stack Overflow. Holding down the command key and typing a the period (⌘ - .) stops Max's scheduler, giving you time to turn off some of the overloading processes. Choose **Resume** to restart Max's timer.

Font Menu

Each Patcher or Text window, and each text object in a Patcher window, can be assigned a specific font and font size.

To change the font of an object, select it, then choose the desired font characteristics from the Font menu. When no objects are selected, choosing a font or font size will set the default font characteristics for new objects created in the active window.

Choosing a font or font size when the Max window is active sets the default font characteristics for any new Patcher or Text window created from then on, and stores these settings in the Max Preferences file.

9-36 — Specify the font size. Font sizes available to your System will be shown in outline.

(Fonts) — The fonts shown in the menu depend on the fonts available in your System file.

Windows Menu

Hide Subwindows — Double-clicking on a table or patcher object opens a *subwindow* to display the contents of the object. The **Hide Subwindows** command closes any subwindows that may be open.

Send to Back — Send the currently active window to the background, behind all other open windows.

Max — Bring the Max window to the front, or show it if it has been hidden.

(other) — The names of all other open windows are listed in the menu. Choosing a window name brings that window to the front, making it the active window. If a window has a diamond to the left of its name, it has been modified since it was opened.

Options Menu

Settings for all the commands in the Options menu are stored in the Max Preferences file, and are recalled each time the Max application is opened.

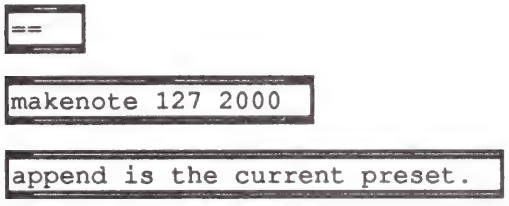
Overdrive — When enabled, **Overdrive** gives priority to music-playing and timing over screen drawing and user interface tasks such as responding to mouse clicks.

All Windows Active — Allows any window to respond to the mouse, without first being brought to the front. Normally (when **All Windows Active** is not enabled), only objects in the frontmost window can respond to mouse clicks. Regardless of the setting of All Windows Active, keyboard events are sent to the front window.

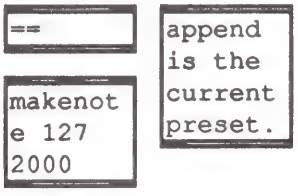
New Object List — When **New Object List** is enabled, Max shows a complete list of available objects whenever you create a new object box. To temporarily override seeing the New Object List window, hold down the Option key when placing a new object box in the Patcher window. See the *Objects* topic for details about the New Object List window.

Auto Fix Width

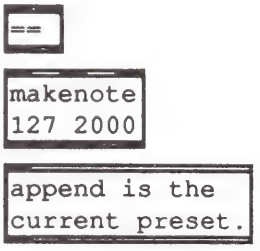
When **Auto Fix Width** is enabled, the width of an object or message box or is automatically adjusted each time the you edit the text inside it. The box is changed so that all the text fits on a single line. You can do this to any selected text box manually by choosing **Fix Width** from the Max menu. Some examples of **Fix Width** are shown below.



Auto Fix Width on



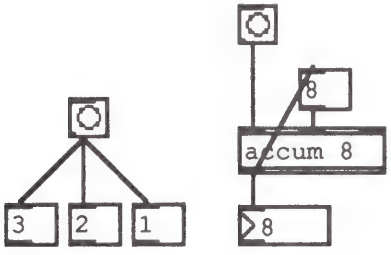
Auto Fix Width off



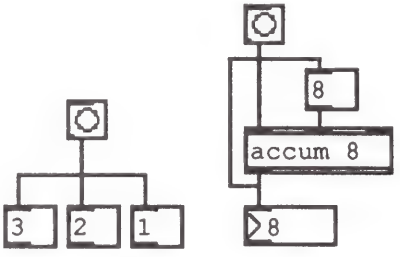
Width fixed by hand

Segmented Patch Cords

When **Segmented Patch Cords** is *not* checked, a patch cord is drawn between two objects simply by dragging with the mouse from the outlet of one object to the inlet of another object.



Straight patch cords



Segmented patch cords

The method of drawing patch cords to connect objects differs slightly when **Segmented Patch Cords** is checked. First click on the outlet of one object, then click at each of the points where you want the patch cord to bend, then click on the inlet of the other object. The shape of a segmented patch cord can be adjusted afterward by dragging on one of the segments with the mouse.

If you make a mistake while drawing a segmented patch cord, you can get rid of the most recent segment by option-clicking with the mouse. Or you can get rid of the whole patch cord by command-clicking anywhere.

Although segmented patch cords redraw more slowly when selected or deselected, they function exactly like straight patch cords, and do not take up additional memory when the patch is loaded.

Assistance

When **Assistance** is enabled, the Patcher window gives you a running commentary about the function of each object's inlets and outlets when the mouse

moves over an inlet or outlet. An example of Assistance for the **accum** object is shown below.

When you edit a patch to be used as an object inside other patches, you can type in your own Assistance messages for your object's inlets and outlets. Select an **inlet** or **outlet** object in your patch, and choose **Get Info...** from the Max menu. You will be presented with a dialog box which lets you type in a description of that **inlet** or **outlet**. When you use your object inside another patch, the assistance messages will appear when the mouse is placed on the object box's inlets or outlets.

See Also

Objects	Creating a new object
menu	Pop-up menu, to display text and send messages

MIDI Messages

A MIDI message is comprised of a *status byte*, which identifies the message type, followed in most cases by one or more *data bytes*. The most significant bit of data bytes is always 0, to distinguish them from status bytes, whose most significant bit is 1. Except for System Exclusive messages, the status byte specifies exactly how many data bytes will follow it. System Exclusive messages are used for synthesizer patch dumps and parameter changes.

MIDI Objects

Max MIDI objects extract the essential data from incoming messages, so you don't need to know the details of the structure of MIDI messages. These objects are listed in the table below describing each type of MIDI message.

Raw MIDI

If you wish to deal with receiving and transmitting entire MIDI messages yourself, you can use the **midiin** and **midiout** objects. The **midiparse** and **midiformat** objects filter and format raw MIDI, and **sxformat** can help in formatting system exclusive commands for transmission by **midiout**.

To help you in managing MIDI data in Max, we have provided two reference charts here. The first chart identifies the different types of MIDI messages and shows their format. The second chart identifies controller numbers (the second byte of a MIDI control change message) which have been assigned a specific function.

MIDI Messages

Channel Messages						
Channel Messages use the lower 4 bits of the status byte to indicate the MIDI channel of the message. 0 is MIDI channel 1, and 15 is MIDI channel 16.						
Function	*Objects*	*Status Byte*			*2nd Byte* (0-127)	*3rd Byte* (0-127)
		Decimal	Hex	Binary		
Note Off	**xnotein, xnoteout**	128-143	80-8F	1000xxxx	Key Number	Release Velocity
Note On	**notein, noteout**	144-159	90-9F	1001xxxx	Key Number	Velocity
Poly Pressure	**polyin, polyout**	160-175	A0-AF	1010xxxx	Key Number	Aftertouch
Control Change	**ctlin, ctlout**	176-191	B0-BF	1011xxxx	Controller Number	Controller Data
Program Change	**pgmin, pgmout**	192-207	C0-CF	1100xxxx	Program Number	
Aftertouch	**touchin, touchout**	208-223	D0-DF	1101xxxx	Aftertouch Value	
Pitch Bend	**bendin, bendout**	224-239	E0-EF	1110xxxx	Bend (LSB)	Bend (MSB)
System Messages						
System Exclusive	**sysexin, midiout**	240	F0	11110000	Mfgr ID Number	Arbitrary
Song Pos Ptr	**midiin, midiout**	242	F2	11110010	Position (LSB)	Position (MSB)
Song Select	**midiin, midiout**	243	F3	11110011	Song Number	
Tune Request	**midiin, midiout**	246	F6	11110110		
End of Sys Ex	**sysexin, midiout**	247	F7	11110111		
Clock	**rtin, midiout**	248	F8	11111000		
Start	**rtin, midiout**	250	FA	11111010		
Continue	**rtin, midiout**	251	FB	11111011		
Stop	**rtin, midiout**	252	FC	11111100		
Active Sensing	**midiin, midiout**	254	FE	11111110		
System Reset	**midiin, midiout**	255	FF	11111111		

Control Changes

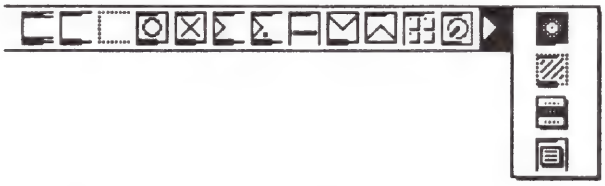
Controller Numbers		
Function	*Ctl Num*	*Values*
Continuous Controllers (MSB)	**0-31**	
Modulation Wheel	1	0-127
Breath Controller	2	0-127
Foot Controller	4	0-127
Portamento Time	5	0-127
Data Entry	6	0-127
Main Volume	7	0-127
Extra precision for the above (LSB)	**32-63**	
On/Off Switch Controllers	**64-95**	
Sustain Pedal	64	127 and 0
Portamento On/Off	65	127 and 0
Sostenuto Pedal	66	127 and 0
Soft Pedal	67	127 and 0
Other	**96-121**	
Data Entry Yes (+1)	96	127
Data Entry No (-1)	97	127
Channel Mode Messages	**122-127**	
Local Control On/Off	122	127 and 0
All Notes Off	123	0
Omni Mode Off	124	0
Omni Mode On	125	0
Mono On	126	0
Poly On	127	0

Objects

Creating a new object in the Patcher

Object Palette

When the Patcher window is unlocked, the *object palette* appears at the top of the window. To create a new object in the Patcher window, simply click on the desired tool in the object palette (the cursor will change to that tool), then click where you want to place the object.



When the Patcher window is narrower than the width of the icons in the palette, the far right palette item in the window becomes an arrow. Holding down the mouse button on the arrow causes the rest of the object palette to be displayed as a hierarchical menu. Choose the desired object from the menu. Note that as you click on different objects in the palette, they are described in the Assistance area of the Patcher window.

If you choose the wrong object by mistake, you can cancel by clicking on the blank area at the far left of the palette or pressing the Delete key.

Directory of Palette Objects

Built-in Objects

- Object Box
- Message Box
- Button
- Toggle
- int Number Box
- float Number Box
- Slider
- Inlet
- Outlet
- Preset

External Objects

- Dial
- Graphic Switch
- Graphic Gate
- Horizontal Slider
- Keyboard Slider
- LED
- Transparent Button
- Pop-up Menu
- Vertical Slider

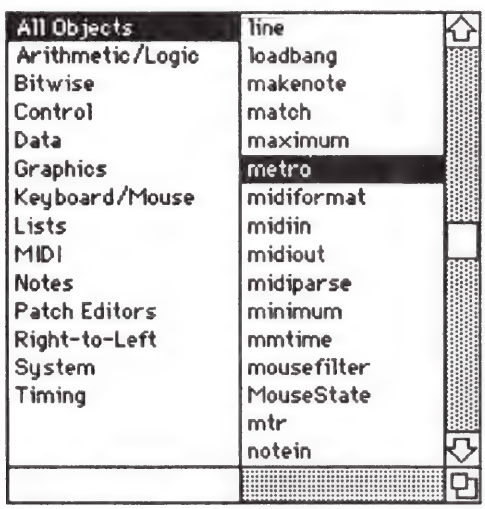
Object Box

The most important graphical object in the palette is the first item, the *Object Box*. The function of an object box depends on what you type into it, for example metro 100.

You can type names of built-in or external Max objects, or names of Max documents you have saved in directories specified in the File Preferences dialog. Note that in order to be typed into an object box, the name of a Max document must not contain any spaces or else be surrounded in "smart quotes" (which are typed as option-[and option-shift-[).

New Object List

Because there are so many names which may be typed into an object box, you might need some help remembering all the possible choices. When **New Object List** is checked in the Options menu, Max provides you with a complete list of all available object names each time you place a new object box in the Patcher window.



New Object List window

The New Object List window has two columns. The left column lists categories of objects, and the right section list the names of objects in the selected category. The *All Objects* category lists all object names known to Max.

To find an object name, you can scroll through the list, type the first few letters of the desired object name, or use the up and down arrow keys on the Macintosh keyboard.

To choose an object name from the list and place it in the object box, double-click on a name, or press Return, Enter, or Space to choose the highlighted name. When an object name has been chosen, the New Object List disappears. The blinking insertion point remains inside the object box, allowing you to type in any arguments the object may require. Finally, click anywhere outside of the object box, or press the Enter key, and the new object will be created.

When you are in the New Object List window, you can get help on the selected object name by holding down the option key while you double-click, press Return, Enter, or Space.

If you want the New Object List to disappear to type in an object name yourself, press the Delete (Backspace) key, or click anywhere outside the New Object List. The blinking insertion point will be placed back inside the object box, so that you can type in the name. To create a new object without even seeing the New Object List, hold down the Option key as you place the object box in the Patcher window, or uncheck **New Object List** in the Options menu.

Max Object List

The Max Object List is a text file you create and save under the name *Max Object List*, in the same folder as the Max application. The Max Object List lets you customize the names in the New Object List, adding your own names you might type often. The objects you add can be external Max objects, or they can be your own patches you have created and saved as Max documents.

The Max Text window is ideal for creating or editing the Max Object List file, since it saves documents in TEXT format. Other word processors such as MacWrite and Microsoft Word, do not automatically save documents in this format.

The proper format for the text in the Max Object List is as follows: on each line, type the name of the object category, a tab stop, and the name of the object. For example:

MIDI	xbendin
MIDI	xbendout
Arithmetic/Logic	minimum
Arithmetic/Logic	maximum
Notes	buildSeq

Note that buildSeq is not the name of an existing Max external object, but could be a patch that you have written and want to use as an object.

After editing the Max Object List file, you need to restart Max before it will take effect.

Note that the category name must be one of those already used in the New Object List window, but the object name can be any single word which does not begin with a number and does not contain a comma or semicolon.

The Max Object List is handy for reducing the number of objects in your startup folder. Place only graphical external objects such as **hslider** and **kslider** into the startup folder, and place all other externals into a folder specified in the File Preferences dialog. Then add these names to the Max Object List, so they can be typed into an object box more quickly.

About OMS

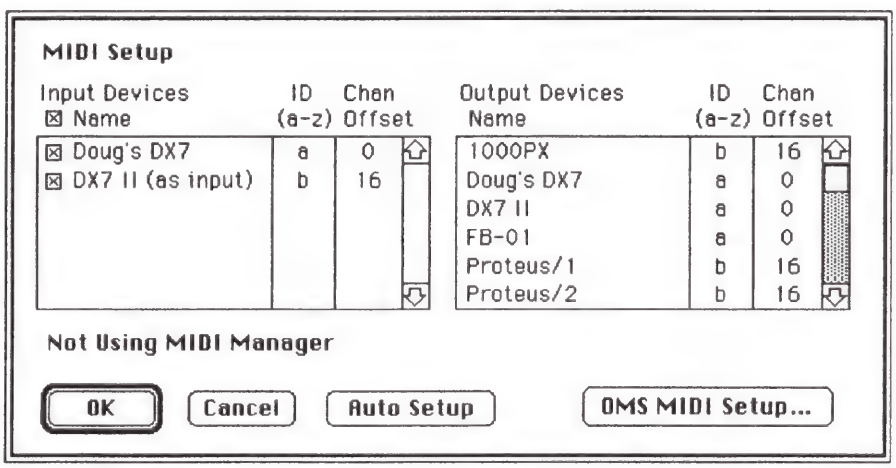
OMS provides a connection between Max and MIDI devices. Without OMS, Max can only address MIDI devices directly connected to the two serial ports. OMS provides a device-independent way for Max to send and receive MIDI from any device which has an OMS driver, including multi-port MIDI interfaces and hardware devices such as the Digidesign SampleCell. Devices are addressed by *name* in OMS, not port letter or MIDI channel number. For example, if you have a DX7 connected to your modem port, you could have an OMS "output port" called DX7 and an OMS "input port" called DX7. OMS can also be used in conjunction with the MIDI Manager, in which case the Apple MIDI Driver is replaced with the OMS MIDI Manager driver.

OMS consists of an INIT, the OMS MIDI Manager driver, and a folder containing device drivers and setup files. You install OMS by placing all of these files in your System Folder. In addition, the OMS Setup application allows you to name the devices connected to the physical ports of your computer. Once you've done this, all OMS-compatible applications will refer to ports and devices by these names.

If OMS has been installed, Max will automatically use the OMS MIDI drivers instead of its own drivers. A setting shared by all OMS-compatible applications determines whether the MIDI Manager is used or not. This setting can be changed in the OMS Setup application, or by clicking OMS Midi Setup... in the Midi Setup dialog which appears when Max is running under OMS.

Using the Midi Setup Dialog in OMS

Since you address multiple MIDI ports in Max with letters a-z or MIDI channel numbers above 16, a certain amount of translation is necessary between OMS device names and the Max way of doing things. You match OMS device names with letter abbreviations and ranges of MIDI channels with the Midi Setup dialog. This dialog will appear the first time you run Max under OMS, and can also be accessed by choosing **Midi Setup...** from the File menu.



To assign an abbreviation and MIDI channel range to a device, use the Numericals next to each device name. The Channel Offset sets the number added to a MIDI channel for identifying an input device or specifying an output device. For example, if an device has a Channel Offset of 32,

sending the number 33 into a channel inlet of a MIDI output object will specify MIDI channel 1 on this device.

Note that output devices which are connected to the same *physical* interface will share the same abbreviation and channel offset.

If you have multiple OMS output devices set to the same MIDI channel offset, Max will *not* send MIDI data out on all the devices. Rather, it will use the first one in the list (at the time the MIDI output object was created). However, a MIDI input object which is not tied to a specific port (i.e. one which does not have an OMS device name or abbreviation as an argument) will "merge" all *input* devices set to the same MIDI channel offset. In this case, there would be no way for a Max application to distinguish the source of input to these objects based on MIDI channel, which is not necessarily a bad thing (you may wish to treat all MIDI input identically, no matter where it comes from).

If multiple devices share the same abbreviation, Max will use the first one in the list at the time a MIDI input or output object is created with that abbreviation as an argument. Changing the abbreviations of devices has no effect on already existing objects, although it will have an effect on the meaning of port messages sent to MIDI output objects with abbreviations as argument.

Click the **Auto Setup** button to have Max arrange a "standard" set of abbreviations and channel offsets. Typically, this assigns the first letters of the alphabet and lower channel offsets to the first device names listed.

If you are using the MIDI Manager, six additional input and output devices will be shown. Named *MIDI Mgr Input 1* to *MIDI Mgr Input 6* and *MIDI Mgr Output 1* to *MIDI Mgr Output 6*, these correspond to Max's MIDI Manager "virtual port" connections. When using OMS, these MIDI Manager ports are available *in addition* to the "direct" connections with your MIDI interface or other devices provided through OMS. When in OMS, you will use the MIDI Manager ports for sending MIDI data from Max to another Macintosh application, or from another application to Max.

Clicking the **OMS MIDI Setup...** button opens the standard OMS MIDI Setup dialog, which allows you to select which serial ports are used for OMS and the Compatibility Mode, one of three possibilities:

- Use MIDI Manager
- Only OMS Applications (allows OMS applications to send and receive MIDI in the background)
- Allow non-OMS applications (relinquishes the serial port when OMS applications are in the background)

If you use the OMS MIDI Setup dialog to stop using the MIDI Manager, the Max MIDI Manager input and output ports will disappear from the list of device names in the main Midi Setup dialog box.

MIDI Manager

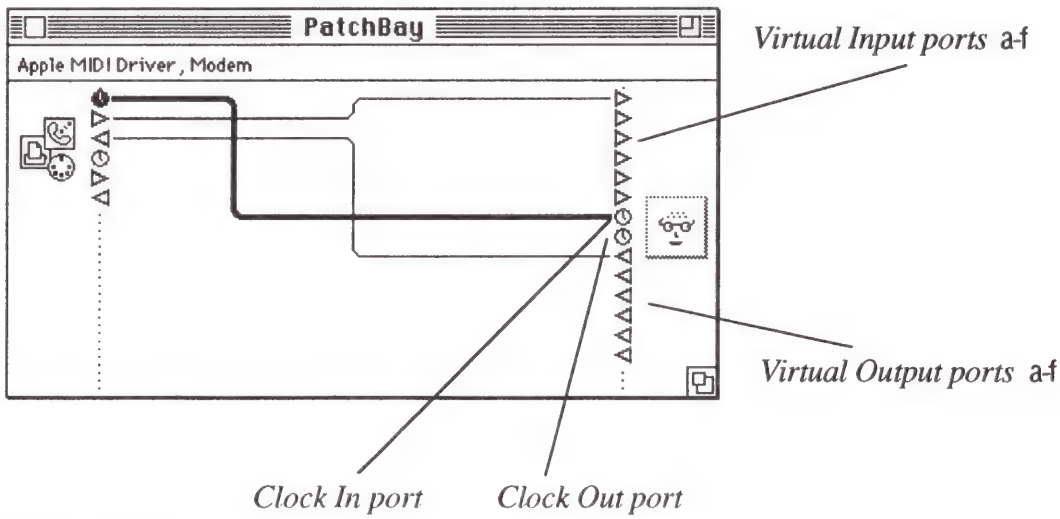
The MIDI Manager is an INIT—a program loaded into memory when you start your computer—which lets you use the Apple MIDI Driver. When the MIDI Manager is being used, Max transmits MIDI through six *virtual* input and output ports which can be connected to the serial ports or other applications using the Patch Bay application. When the MIDI Manager is not being used, Max uses the two serial ports directly. Choose **Midi Setup...** from the File menu to configure whether Max is using the MIDI Manager or talking directly to the serial ports.

Using PatchBay

You use the MIDI Manager's PatchBay application to connect applications to the serial ports and other applications. An application's virtual ports are designated by arrows next to the program's icon.

The Apple MIDI Driver has one input arrow and one output arrow for each of the serial ports. Max has six input arrows and six output arrows (lettered a-f in Max). To specify a connection, just drag from an output arrow (say, the output arrow of the modem port in the Apple MIDI Driver) to an input arrow (say, input port a of Max). In the default configuration, Max's a input and output ports are connected to the output and input arrows of the modem port. Any of Max's ports can be connected to either serial port, or to the ports of other MIDI applications.

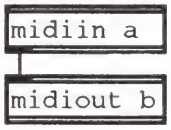
If you want Max to receive time code from the MIDI Manager, connect the Apple MIDI Driver's clock icon to Max's Clock In icon as shown below.



Max's MIDI Objects

Each object in Max that transmits or receives MIDI data can be set to communicate with a specific port by typing in a letter argument after the name of the object. When the MIDI Manager is not being used, a specifies the modem port and b specifies the printer port. Any other letter is an

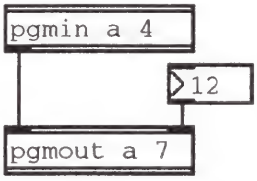
invalid port argument. When the MIDI Manager is being used, however, any letter from a to f can be used, specifying a virtual port of the MIDI Manager.



Receive only from port a

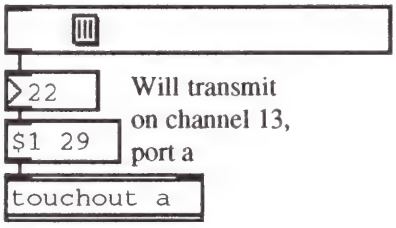
Send only out port b

For MIDI objects that receive and transmit channel voice messages, such as **notein** and **bendout**, a letter argument followed by a number argument indicates a port and a specific MIDI channel on which to receive or transmit MIDI messages. For transmitting objects (except **midiout**), channel numbers are received in the rightmost inlet to change the channel.



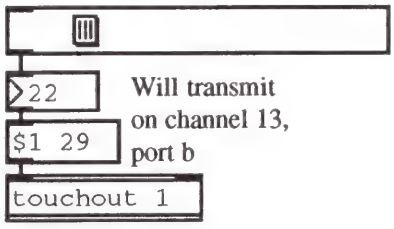
Channel number can be changed by a number received in the right inlet

A number alone can be used in place of a letter and number combination. Numbers 1-16 specify a channel number on port a, numbers 17-32 specify channels 1-16 on port b, etc. Numbers greater than 16 can be received in the right inlet, to specify both port and channel. When a letter argument is present, however, the port is permanently fixed, and channel numbers greater than 16 received in the right inlet are *wrapped around* to stay within the 1-16 range.



Will transmit on channel 13, port a

Letter argument transmits to only one port.



Will transmit on channel 13, port b

Otherwise, number specifies both port and channel.

MIDI Manager

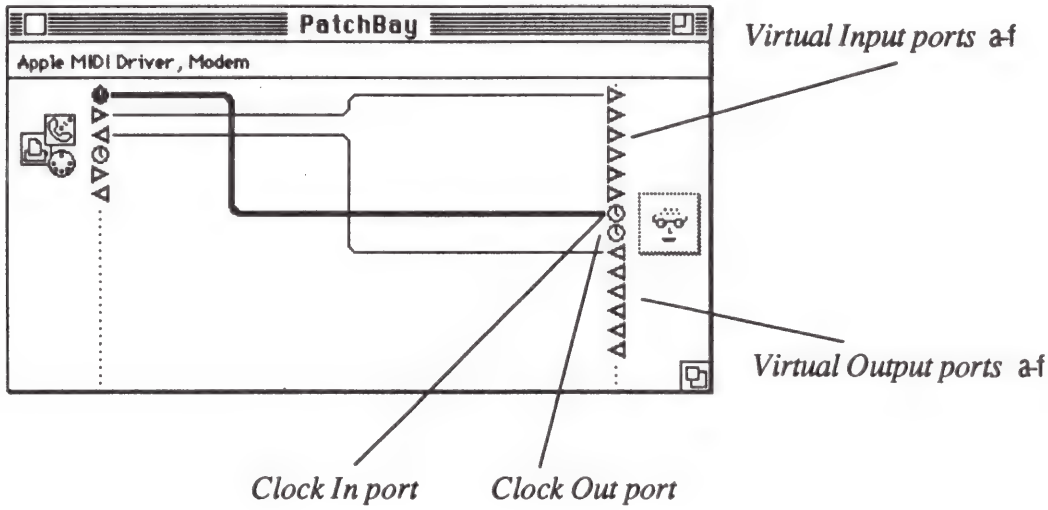
The MIDI Manager is an INIT—a program loaded into memory when you start your computer—which lets you use the Apple MIDI Driver. When the MIDI Manager is being used, Max transmits MIDI through six *virtual* input and output ports which can be connected to the serial ports or other applications using the Patch Bay application. When the MIDI Manager is not being used, Max uses the two serial ports directly. Choose **Midi Setup...** from the File menu to configure whether Max is using the MIDI Manager or talking directly to the serial ports.

Using PatchBay

You use the MIDI Manager's PatchBay application to connect applications to the serial ports and other applications. An application's virtual ports are designated by arrows next to the program's icon.

The Apple MIDI Driver has one input arrow and one output arrow for each of the serial ports. Max has six input arrows and six output arrows (lettered a-f in Max). To specify a connection, just drag from an output arrow (say, the output arrow of the modem port in the Apple MIDI Driver) to an input arrow (say, input port a of Max). In the default configuration, Max's a input and output ports are connected to the output and input arrows of the modem port. Any of Max's ports can be connected to either serial port, or to the ports of other MIDI applications.

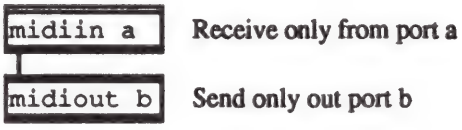
If you want Max to receive time code from the MIDI Manager, connect the Apple MIDI Driver's clock icon to Max's Clock In icon as shown below.



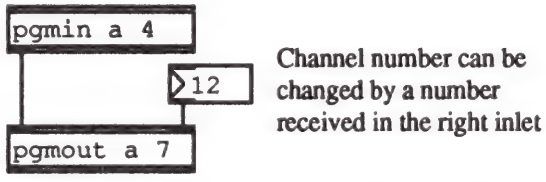
Max's MIDI Objects

Each object in Max that transmits or receives MIDI data can be set to communicate with a specific port by typing in a letter argument after the name of the object. When the MIDI Manager is not being used, a specifies the modem port and b specifies the printer port. Any other letter is an

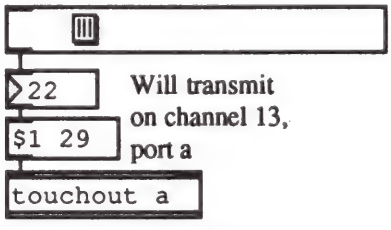
invalid port argument. When the MIDI Manager is being used, however, any letter from a to f can be used, specifying a virtual port of the MIDI Manager.



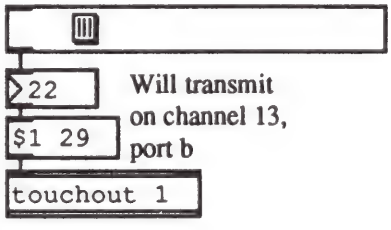
For MIDI objects that receive and transmit channel voice messages, such as **notein** and **bendout**, a letter argument followed by a number argument indicates a port and a specific MIDI channel on which to receive or transmit MIDI messages. For transmitting objects (except **midiout**), channel numbers are received in the rightmost inlet to change the channel.



A number alone can be used in place of a letter and number combination. Numbers 1-16 specify a channel number on port a, numbers 17-32 specify channels 1-16 on port b, etc. Numbers greater than 16 can be received in the right inlet, to specify both port and channel. When a letter argument is present, however, the port is permanently fixed, and channel numbers greater than 16 received in the right inlet are *wrapped around* to stay within the 1-16 range.



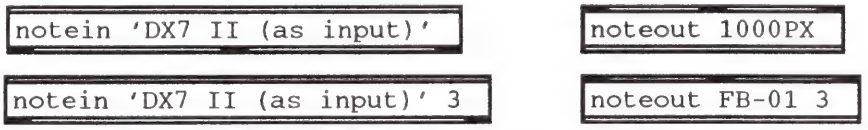
Letter argument transmits to only one port.



Otherwise, number specifies both port and channel.

Specifying Ports in OMS

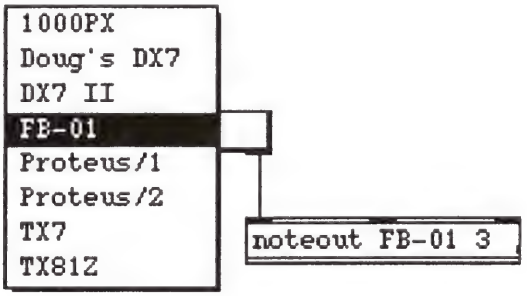
When using OMS, MIDI ports may be specified three ways: by device name, letter abbreviation (a to z), or MIDI channel number. Device names are used as arguments to MIDI objects in place of letters. Here are some examples. Note the use of "smart" single quotes around names of devices which contain spaces.



Refer to the OMS Topic for information about assigning abbreviations and MIDI channel ranges in OMS.

The port Message

All MIDI output objects accept a message that changes the port they use to send MIDI data. The port message takes one argument, which is the letter or (when using OMS) device name associated with the port. The word port can be omitted, which is convenient for making a pop-up menu with port names or abbreviations, as shown below.



After receiving the port message, the MIDI output object acts as if the port had been specified as an argument—all MIDI channels above 16 wrap around to 1-16, and data is always sent on the specified port.

This page intentionally left blank

Punctuation

Punctuation in Object Boxes

Many non-alphabetic characters have a special meaning in Max when included in objects and messages.

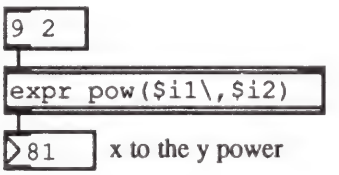
Many characters are object names in their own right, signifying arithmetic, relational, and bitwise *operators* for numerical calcualtions. These object names are **+**, **-**, *****, **/**, **%** (arithmetic operators), **<**, **<=**, **==**, **!=**, **>=**, **>**, **&&**, **||** (relational operators), and **&**, **|**, **<<**, **>>** (bitwise operators). See the descriptions for these objects at the end of the *Objects* section for more information.

The dollar sign ($) and the pound sign (#) are used in object boxes to indicate *changeable arguments*. A changeable argument is replaced by a value supplied either in the inlet (in the case of $) or as typed-in arguments to a patch that contains the object (in the case of #). The *Arguments* topic has detailed information about $ and # in object boxes.

The semicolon (;) is used to indicate the end of a message, and is not allowed in object boxes. Semicolons are a way of forcing a carriage return in a **comment** object.

The semicolon used to indicate the end of a line in text files containing the contents of **coll**, **mtr**, and **seq** objects and in text files which contain a script for the **llb** object .

A comma (,) is generally another character to avoid using in object boxes, but may be used in an **expr** or **if** object, to separate objects within a function in a mathematical expression, as in the example below. Note that a comma in an object box should always be preceded by a backslash (\), so that Max does not try to interpret it as a special character.



Use a backslash when you want to use a special character, but don't want Max to interpret it as such. In the example above, the comma is needed to separate arguments to the pow function.

Punctuation in a Message Box

The dollar sign ($) can be used in a message box to indicate a changeble argument. When the message box contains a $ and a number (such as $2) as one of its arguments, that argument will be replaced by the corresponding argument in the incoming message before the message box sends out its own message.

The pound sign, followed by a number (such as #2), in a message has the same meaning as in an argument of an object box. When the patch containing a # argument is used as a subpatch inside another Patcher, the # argument is replaced by the corresponding argument typed into the subpatch object box in the main Patcher. See the *Arguments* topic.

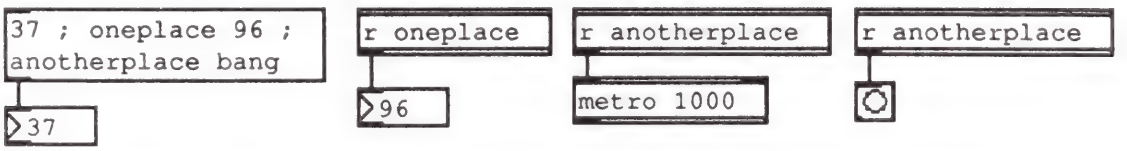
Punctuation

Special characters in objects and messages

A comma (,) in a message box is used to send a series of separate messages. The comma indicates the end of one message and the beginning of the next message.

In the above example, the message box on the left sends out a single message, 60 64 1 as a list. The message box on the right sends out three separate messages—first 144, then 60, then 64.

A semicolon (;) in a message box is used to send messages to remote **receive** objects. When a semicolon is present in a message box, the first item after the semicolon is a symbol indicating the name of a **receive** object, and the rest of the message (or up to the next semicolon) is sent to all **receive** objects with that name, rather than out the message box's outlet.



As in an object box, the backslash (\) in a message negates the special characteristics of the character immediately it preceds

The number-letter combination 0x (zero-x) allows numbers to be typed into object and message boxes hexadecimal form (useful for people who think of MIDI bytes in hex). For example, the message 0x9F 0x3C 0x40 is equivalent to the message 159 60 64.

See Also

Arguments $ and #, changeable arguments to objects

Using a table for probability distribution

Quantile

The quantile message

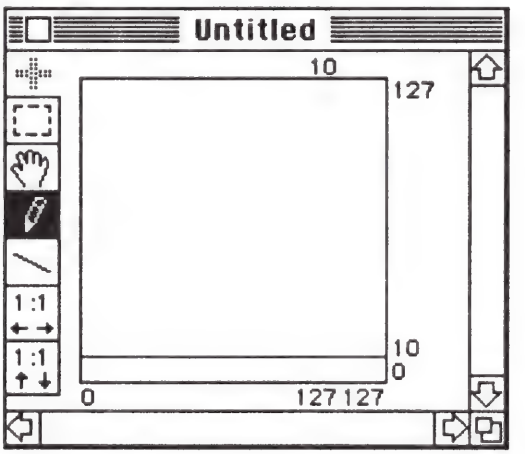
One of the messages understood by the **table** object is the word quantile, followed by a number. If you have read the description of this message, under **table**, you may have wondered what utility this complicated calculation might have. This section provides some examples. Here is the description of what quantile does.

quantile — In left inlet: The word quantile, followed by a number, multiplies the number by the sum of all the values in the **table**. This result is then divided by 2^{15} (32,768). Then, **table** sends out the address at which the sum of all values *up to* that address is greater than or equal to the result.

As the argument of the quantile message progresses from 0 to 32,768, each address in the **table** occupies a portion (*quantile*) of the 0 to 32,768 range, proportional to the "weight" given by the value stored at that address. Repeated quantile messages using random numbers cause each address to be sent out with a frequency roughly proportional to the value at that address.

Examples

Suppose we have a **table** of 128 numbers, all set to 10.



Here are the results of some quantile messages on this **table**. Note that the total sum is 128 * 10 or 1280.

quantile 0 — Always causes an output of 0.

quantile 16384 — Returns the index up to which the sum of the values is half of the total sum. In this case, this would be 63, since 64 * 10 = 640 which is half of 1280.

bang — A bang is equivalent to a quantile message with a random number between 0 and 32768 as its argument. Repeated bangs to a **table** will return **table** indices which contain higher values more often than indices which contain with lower values. In the quantile example above, all indices are equally likely to be returned by bang,

because all the values in the **table** are the same. However, if one of the values were 1000, the index at which the value was 1000 would occur far more frequently than any other **table** index. Exactly how frequently? This is determined by first taking the sum of all values in the **table**, which, for a **table** with 127 indices set to 10 and one at 1000 would be 2270. For the one index set to 1000, we divide 1000 by the sum 2270 and get a probability of 44 percent. For any of the other 127 indices set to 10, the probability is .44 percent that any one will be chosen. So, the index set to 1000 will occur about 100 times more frequently than an index set to 10.

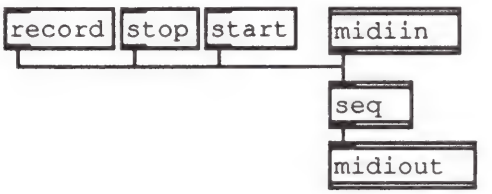
See Also

Histo	Make a histogram of the numbers received
table	Store and edit an array of numbers
Tutorial 33	Probability tables

seq

Max has three objects for recording and playing back performances: **seq**, **follow**, and **mtr**. The "performance" can come from outside Max—from a MIDI controller, or another MIDI application via the MIDI Manager—or can be generated algorithmically within Max.

The basic sequencer in Max is **seq**, which records raw MIDI data received in its inlet from **midiin** or **midiformat**, and can play the data back at any speed. The recording and playback process is controlled with messages such as record, start, and stop.



Sequences recorded by **seq** can be written into a separate file to be used again later (**seq** saves the recorded MIDI data in a standard MIDI file format). When **seq** receives a write message, it calls up the standard **Save As...** dialog box. If the file is saved as text (by checking the *Save As Text* option in the **Save As...** dialog), it can be edited by hand by choosing **Open As Text...** from the File menu. MIDI files can also be read into **seq** with a read message.

follow

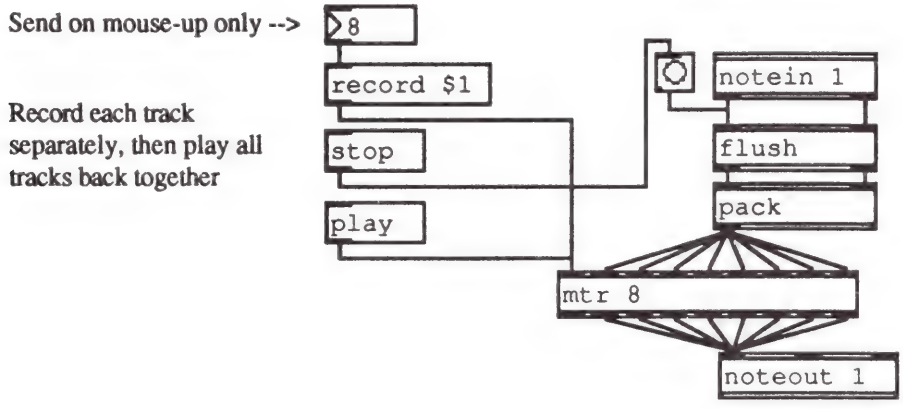
The **follow** object functions exactly like **seq**, but has the added ability to compare a live performance to the performance it has recorded earlier. **follow** can record not only raw MIDI data, but also individual numbers such as note-on pitch values. You can step through the set of recorded notes (or numbers) using the next message. Most interestingly, **follow** contains a score following algorithm, activated by the follow message. follow will compare incoming numbers to those stored in its recorded sequence. If an incoming number matches the next number in the recorded sequence (or a nearby number, just in case the live performer makes a mistake), **follow** reports the index of the matched note. The index can then be used to read other numbers from a **table** or **coll** (providing an accompaniment to the live performer), or can be used to trigger any other process.

mtr

The **mtr** object is a multi-track sequencer which can record up to 32 individual tracks of numbers, lists of numbers, or symbols. With such versatility, it is easy to record not only MIDI events, but a wide variety of other messages. Tracks can be recorded, played, or stepped using the next message, either individually or collectively, and some tracks can be muted while other tracks continue to play. The contents of **mtr** can be written to and read in from separate files, either as individual tracks or as an entire set of tracks.

Sequencing

Recording and playing back MIDI performances



Sample patch using **mtr**

For editing complete MIDI messages as text, **seq** is perhaps more appropriate since it arranges raw MIDI data into a standard MIDI file format. However, raw MIDI data can be filtered with **midiparse** before being sent to **mtr**. Also, a sequence recorded in **seq** can easily be cut out and pasted into an **mtr** file, using Max's text editor.

See Also

follow	Compare a live performance to a recorded performance
mtr	Multi-track sequencer
seq	Sequencer for recording and playing MIDI
Tutorial 35	Sequencing

Shortcuts

Locked Patcher Window

- Option-clicking on any object's box opens a help file for that object
- Command-clicking in any white space unlocks the Patcher window (if it's editable).
- Option-clicking on a window's close box closes all windows except the Max window.
- Option-clicking on the title bar of a subpatch window pops up a menu which allows you to bring to the front any parent windows of the subpatcher. If the subpatcher is an "edit-only" Patcher window (produced by double-clicking on a Patcher object which was read from a Max document), the top item in the list opens the Max document for editing.
- Typing command-period (.) stops the Max scheduler, allowing you to recover from a runaway process which might be taking up too much CPU time to stop by normal methods. After you have remedied the situation which caused the process to get out of hand, choose **Resume** from the Max menu to restart the scheduler.

Unlocked Patcher Window

- Option-clicking on any object's box opens a help file for that object.
- Shift-clicking on an object box reverses the selected state of the object without changing the selected state of other objects.
- Shift-dragging an object box "constrains" the dragging of the object in the horizontal or vertical dimension.
- The arrow keys move the selected boxes by 1 pixel in any direction.
- Option-dragging a rectangle around a set of objects and patch cords selects both the patch cords and the objects.
- To drag a text object which has been selected for editing, move the cursor to the top or bottom edge of the box, then drag, as shown below. This is a handy way to move an object after duplicating it.

Click at the top or bottom edge of a selected text box to drag it

- Command-clicking on any user interface object, such as a **slider** or **Number box**, operates the object as if the Patcher window were locked.

Shortcuts

Power user commands

- Command double-clicking edits objects such as **patcher**, **table**, and **coll** that open when you double-click them in a locked Patcher window.
- Command-clicking in any white space locks the Patcher window.
- Shift-clicking on an outlet of an object uses **Segmented Patch Cords** mode, in which subsequent clicks define the "corners" of the patch cord. If **Segmented Patch Cords** is enabled in the Options menu, shift-clicking uses the normal mode of dragging from outlet to inlet to make a straight patch cord.
- Command-clicking while making a segmented patch cord gets rid of the patch cord and cancels the operation.
- Option-clicking while making a segmented patch cord gets rid of the last segment.
- Option-clicking after selecting the Object Box tool in the palette places object box without showing the New Object List window.
- Option-clicking on an empty box made with the Object Box tool opens the New Object List window.
- Pressing the Delete (Backspace) key after selecting a tool from the palette cancels the operation and returns to the normal cursor.
- Clicking on the white area at the far left of the palette also cancels the selected palette tool.

New Object List

- Delete (Backspace) hides the New Object List window.
- The Space bar enters the text of the selected item in the New Object List into the object box and adds a space afterwards for typing in any arguments.
- Return or Enter enters the text of the selected item in the New Object List into the object box.
- The Up and Down Arrow Keys scroll the selected item up and down.
- Tab switches which column of items is affected by typing.
- Holding the option key down while double-clicking or typing Return, Space, or Enter opens a help file on the selected item.

send, receive, and value

- Double-clicking on a **send**, **receive**, or **value** object brings to the front another window which contains a corresponding **send**, **receive**, or **value** object (i.e, one which uses the same name). Double-clicking on the **send**, **receive**, or **value** object in the newly selected window selects

another window which contains a corresponding object. Continued double-clicking visits all such windows in a cyclic fashion.

Table Editing Window

- Command clicking with the *Pencil tool* magnifies the area around where you clicked. If you are at 8:1 x 8:1 zoom, command-clicking returns to 1:1 x 1:1 magnification, or the minimum allowed zoom above 1:1.

Any Window

- Command-clicking in any window while **All Windows Active** is enabled brings that window to the front (if the window is not already in front).

Librarian Window

- Option-double-clicking on the number of a patch reuses the most recently opened editor window on the patch rather than loading a new editor from the disk, saving both time and memory.

Object Get Info... dialogs

- The **Cut**, **Copy**, and **Paste** keyboard shortcuts (Cmd-X, Cmd-C, and Cmd-V) work in most objects' **Get Info...** dialogs, most notably **inlet**, **outlet**, and **menu**.

Table Window

Choosing **Table** from the New menu will open a new table editing window for drawing number values in a two-dimensional graph. Once values have been drawn into the window, they can be saved in a file, to be read by **table** objects in a Patcher. Alternatively, you can create a new **table** object in a Patcher window, and the object's table editing window will open.

The editing window provides a palette of graphic editing tools for drawing in values.

show crosshairs, to aid precision drawing

select a region to cut, copy, clear, or paste

drag the display, to see another part of the table

draw in values freehand with the mouse

draw values in a straight line, from click to click

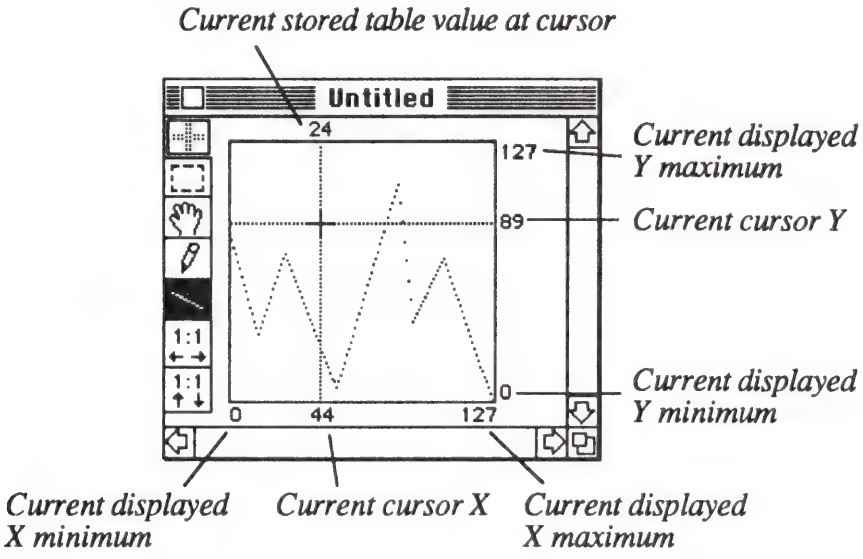
zoom the horizontal display in or out

zoom the vertical display in or out

Values can be drawn by choosing the Pencil tool and dragging inside the graph, or by choosing the Line tool and clicking at points in the graph. The Line tool automatically draws in a straight line, beginning from the previous point you clicked at, and ending at the point you just clicked at.

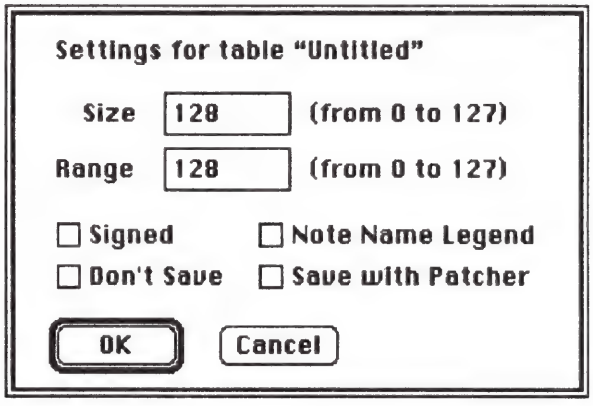
The Selection tool lets you cut, copy, and select regions of the table. Drag across a region of the graph to highlight the region, choose **Cut** or **Copy** from the Edit menu, click at the point where you want to insert what you just cut or copied, then choose **Paste** from the Edit menu. If more than one point is selected, Paste will replace the selection with the values on the Clipboard.

As you move the cursor inside the graph region, a number along the *x* axis at the bottom of the graph shows the cursor's position on that axis, and a number along the *y* axis shows the cursor's current vertical coordinate. The number at the top of the graph shows the *y* value that has been drawn in at the cursor's current *x* value.



Get Info...

Many characteristics of the **table** object and its editing window can be altered by choosing **Get Info...** from the Max menu when the editing window is the active window.



Size determines the number of values in the **table**. A newly created table has 128 values, indexed with numbers from 0 to 127.

Range determines the range of values which can be displayed on the *y* axis of the editing window. A newly created table editing window has a range of 128, from 0 to 127.

If *Signed* is checked, the table window displays negative numbers as well as positive. In effect, the range of displayed values specified by *Range* is doubled when *Signed* is checked, since the range goes in both directions from 0.

If *Note Name Legend* is checked, values are shown on the *y* axis as MIDI note names, rather than numbers.

Once the values have been stored in a file, the file can be read into **table** objects in a Patcher, and the values can be altered. If *Don't Save* is checked, changes made to values in a **table** object will not be saved in the **table**'s separate file. Otherwise, Max will ask if you want to save changes made to the **table**, when the Patcher containing that **table** is closed.

If *Save with Patcher* is checked, the values in a **table** will be saved as part of the patch that contains it.

If you specify an extremely large or small size or range for the table window, you may want to zoom your view of the graph in or out to see the values in a better perspective. Clicking on the zoom arrows in the palette of drawing tools will shrink or enlarge the view of the values in the graph. You can also move about within the table window by "grabbing" the graph with the hand tool, and dragging it until you can see the portion of the graph that you want to edit. Or scroll through the table using the scroll bars.

Using a table

Values you have drawn into a table window can be saved in a file which can be read into **table** objects in a Patcher, by using the filename as the argument to a **table**. A value is sent out of **table**'s left outlet when an address number is received in the left inlet. Values in a **table** object can be modified by messages in its inlets, and the graphic editing window of a **table** can be opened by double-clicking on its object box.

A single file can be shared by many different **table** objects if all the objects have the file's name as an argument.

See Also

Data Structures	Ways of storing data in Max
table	Store and graphically edit an array of numbers
Tutorial 32	The **table** object

Output the absolute value of the input

abs

Input

int — The absolute (non-negative) value of the input is sent out the output.

float — Converted to int, unless **abs** has a float argument.

Arguments

int or float — Optional. Argument with a decimal point forces a float output.

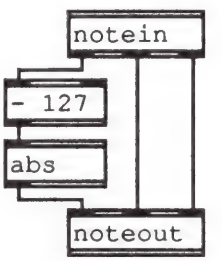
Output

int — The absolute value of the input.

float — Only if there is an argument with a decimal point.

Examples

Output is nonnegative



Used here to invert input

See Also

expr — Evaluate a mathematical expression
Tutorial 14 — Sliders and dials

accum

Store, add to, and multiply a number

Input

int — In left inlet: Replaces the value stored in **accum**, and sends the new value out the outlet.

In middle inlet: The number is added to the stored value, without triggering output.

In right inlet: The stored value is multiplied by the input, without triggering output.

float — In left and middle inlet: Converted to int, unless **accum** has a float argument.

In right inlet: Multiplication is done with floats, even if the value is stored as an int.

bang — In left inlet: Outputs the value currently stored in **accum**.

set — In left inlet: The word set, followed by a number, sets the stored value to that number, without triggering output.

Arguments

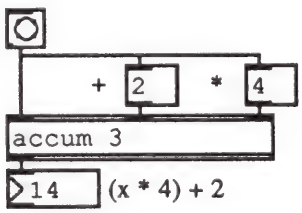
int or float — Optional. Sets the initial value stored in **accum**. An argument with a decimal point causes the value to be stored as a float.

Output

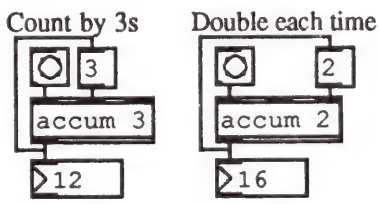
int — The value currently held by **accum**.

float — Only if there is an argument with a decimal point.

Examples



Add to and/or mutiply a stored value



Used here to increment by different amounts

See Also

float — Store a decimal number
int — Store an integer value
Tutorial 21 — Storing numbers

Send 1 when window is active,
0 when window is inactive

active

Input

There are no inlets. Output is triggered automatically when the Patcher window is activated or deactivated.

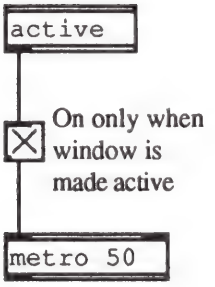
Arguments

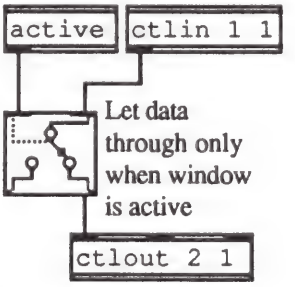
None.

Output

int When the Patcher window that contains **active** is activated, **active** sends out 1. When the window is made inactive, **active** sends out 0.

Examples





Turn on a process or open a gate when the window is made active

See Also

closebang	Send bang when Patcher window is closed
loadbang	Send a bang automatically when patch is loaded
Tutorial 40	Automatic actions

append

Append arguments at the end of a message

Input

set — The word set, followed by any message, will replace the message stored in **append**, without triggering output.

anything else — The message stored in **append** is appended, preceded by a space, to the end of any message that is received in the inlet, and the combined message is sent out the outlet.

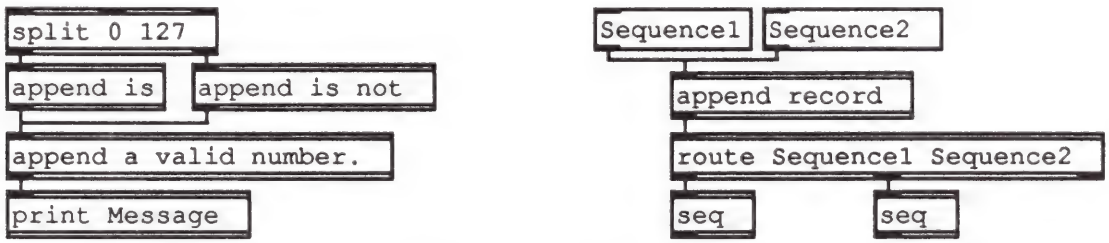
Arguments

anything — Optional. Sets the message that will be appended to the end of incoming messages.

Output

anything — The message received in the inlet is combined with the message stored in **append**, and then sent out the outlet.

Examples



Symbols can be combined into meaningful messages with **append**

See Also

prepend — Put one message at the beginning of another
Tutorial 25 — Managing messages

Store a collection of numbers

bag

Input

int — In left inlet: The number is either added to or deleted from the collection of numbers stored in **bag**, depending on the number in the right inlet.

In right inlet: The number is stored as an indicator of whether to include or delete the next number received in the left inlet. If non-zero, the number received in the left inlet is added to the **bag**. If 0, the number is deleted from the **bag**.

No output is triggered by a number received in either inlet.

float — Converted to int.

bang — In left inlet: Causes **bag** to send all its numbers out the outlet.

clear — In left inlet: Deletes the entire contents of the **bag**.

list — In left inlet: If the second number is not 0, the first number is included in the **bag**. If the second number is 0, the first number is deleted from the **bag**.

send — In left inlet: The word send, followed by the name of a **receive** object, sends the result of a bang message to all **receive** objects with that name, instead of out **bag**'s outlet.

Arguments

any symbol — Optional. Causes **bag** to store duplicate numbers. If there is no argument, **bag** will store only one of each number at a time. The argument must not be a number.

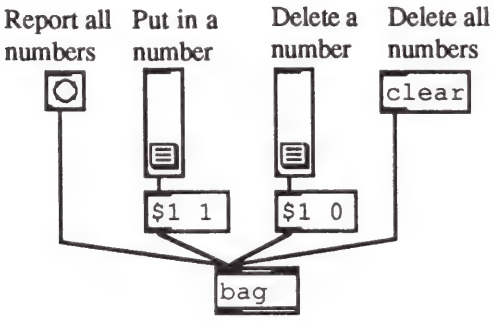
Output

int — The numbers stored in **bag** are sent out one a time, in reverse order from that in which they were stored.

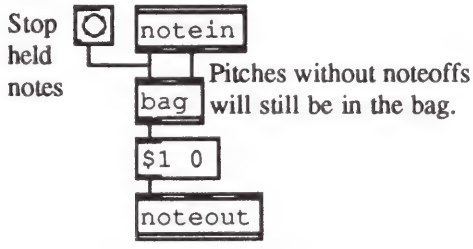
bag

Store a collection of numbers

Examples



Store a collection of numbers



Used here to detect held notes

See Also

coll	Store and edit a collection of different messages
Data Structures	Ways of storing data in Max
funbuff	Store x,y pairs of numbers

Send a bang to many places, in order

bangbang / b

Input

anything — Causes a bang to be sent out all outlets, in right-to-left order.

Arguments

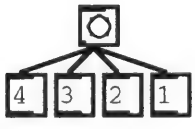
int — Optional. Sets the number of outlets. Limited between 1 and 10. Any number greater than 10 is set to 10; any number less than 1 is set to 2. If there is no argument, there will be 2 outlets.

float — Converted to int.

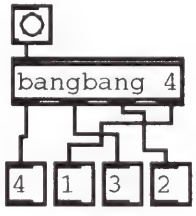
Output

bang — When a message is received in the inlet, bang is sent out each outlet, in order from right to left.

Examples



Order is normally right-to-left



Order is specified by **bangbang**

See Also

button — Flash on any message, send a bang
trigger — Send a number to many places, in order
Tutorial 7 — Right-to-left order

bendin

Output incoming MIDI pitch bend values

Input

(MIDI) **bendin** receives its input from a MIDI pitch bend message received directly from the serial ports, or from the *virtual ports* of the MIDI Manager and/or OMS.

port The word port, followed by a letter a-z or the name of an OMS Input Device, sets the port from which the object receives incoming pitch bend messages. The word port is optional and may be omitted.

Arguments

a-z Optional. Specifies the port from which to receive incoming pitch bend messages. If there is no argument, **bendin** receives from all channels on all ports.

OMS name Optional. When using OMS, you may use the name of an OMS Input Device as an argument to specify a port instead of a-z.

a-z and int A letter and number combination (separated by a space) indicates a port and a specific MIDI channel on which to receive pitch bend messages. Channel numbers greater than 16 will be *wrapped around* to stay within the 1-16 range.

int A number alone can be used in place of a letter and number combination. Numbers 1-16 specify a channel number on port a, numbers 17-32 specify channels 1-16 on port b, etc.

If the MIDI Manager is being used, a letter a-f specifies a virtual port. If the MIDI Manager is not being used, the letter a specifies the modem port and b specifies the printer port. If OMS is being used, the letter a-z can be an abbreviation for any OMS or MIDI Manager port (configured using the **MIDI Setup...** dialog).

Output

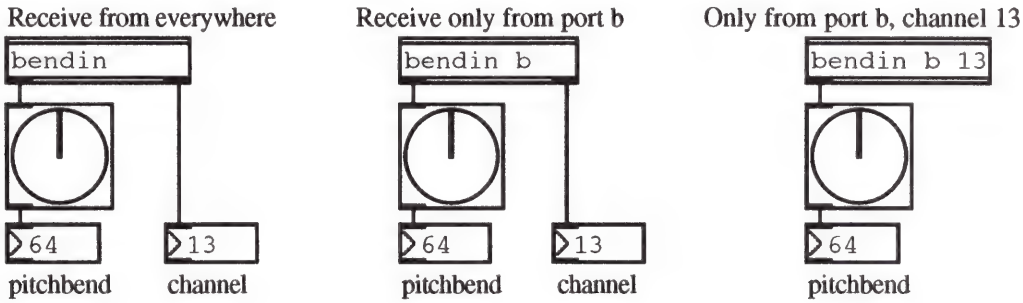
int If a specific channel number is included in the argument, there is only one outlet. The output is the incoming pitch bend value from 0-127 (the most significant byte of the MIDI pitch bend message) on the specified channel and port.

If there is no channel number specified by the argument, **bendin** will have a second outlet, on the right, which will output the channel number of the incoming pitch bend message.

Output incoming MIDI pitch bend values

bendin

Examples



Pitch bend messages can be received from everywhere, a specific port, or a specific port and channel

See Also

bendout	Transmit MIDI pitch bend messages
midiin	Output incoming raw MIDI data
xbendout	Prepare extra precision MIDI pitch bend messages
xbendin	Interpret extra precision MIDI pitch bend values
OMS	Using Max with OMS
Ports	How ports are specified
Tutorial 16	More MIDI ins and outs

bendout

Transmit MIDI pitch bend messages

Input

int — In left inlet: The number is transmitted as a MIDI pitch bend value on the specified channel and port. Numbers are limited between 0 and 127.

In right inlet: The number is stored as the channel number on which to transmit the pitch bend messages.

float — Converted to int.

list — In left inlet: The first number is the pitch bend value, and the second number is the channel, of a MIDI pitch bend message, transmitted on the specified channel and port.

port — In left inlet: When not using OMS, the word port, followed by a letter a-f, specifies the port used to transmit the MIDI messages. When using OMS, port can be followed by a-z or the name of an OMS Output Device. In either case, the word port is optional and can be omitted.

Arguments

a-z — Optional. Specifies the port for transmitting MIDI pitch bend messages. Channel numbers greater than 16 received in the right inlet will be *wrapped around* to stay within the 1-16 range. If there is no argument, **bendout** initially transmits out port a, on MIDI channel 1.

a-z and int — A letter and number combination (separated by a space) indicates a port and a specific MIDI channel on which to transmit pitch bend messages. Channel numbers greater than 16 will be *wrapped around* to stay within the 1-16 range.

OMS name — Optional. When using OMS, the name of an OMS Output Device may be used as the first argument to specify the port.

int — A number alone can be used in place of a letter and number combination. Numbers 1-16 specify a channel number on port a, numbers 17-32 specify channels 1-16 on port b, etc. In this case, channel numbers received in the right inlet will specify both the port and the channel number.

If the MIDI Manager is being used, a letter a-f specifies a virtual port. If the MIDI Manager is not being used, the letter a specifies the modem port and b specifies the printer port. If OMS is being used, the letter a-z can be an abbreviation for any OMS or MIDI Manager port (configured using the **MIDI Setup...** dialog).

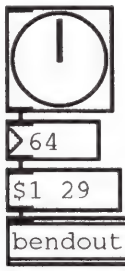
Output

(MIDI) — There are no outlets. The output is a MIDI pitch bend message transmitted directly to the serial ports, or to the *virtual ports* of the MIDI Manager and/or OMS.

Transmit MIDI
pitch bend messages

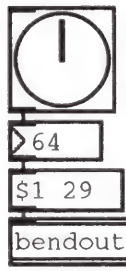
bendout

Examples



Will transmit on channel 13, port a

Letter argument transmits to only one port



Will transmit on channel 13, port b

Otherwise, number specifies both port and channel

See Also

bendin	Ouput incoming MIDI pitch bend values
midiout	Transmit raw MIDI data
xbendout	Prepare extra precision MIDI pitch bend messages
xbendin	Interpret extra precision MIDI pitch bend values
Ports	How ports are specified
Tutorial 16	More MIDI ins and outs

Borax

Report current information about note-ons and note-offs

Input

int — In left inlet: The number is the pitch value of a MIDI note-on message or note-off message (note-on with a velocity of 0). The pitch is paired with the velocity in the middle inlet. **Borax** ignores note-on messages for pitches it is already holding, and ignores note-off messages for pitches that have already been turned off. If the note is not a duplicate, **Borax** sends out the pitch and velocity values, as well as other information.

In middle inlet: The number is stored as the velocity, to be paired with pitch numbers received in the left inlet.

float — In middle inlet: Converted to int.

list — In left inlet: The second number is stored as the velocity, and the first number is used as the pitch, of a pitch-velocity pair. If the note is not a duplicate, **Borax** sends out the pitch and velocity values, as well as other information.

delta — In left inlet: Causes the *delta time* (the time elapsed since the last note-on) and the *delta count* (the number of delta times that have been reported) to be sent out.

bang — In right inlet: Resets **Borax** by sending note-offs for all notes currently being held, erasing **Borax**'s memory of all notes received, and setting its counters and its clock to 0.

Arguments

None.

Output

int — Out left outlet: Each note-on received by **Borax** is assigned a unique number, equal to the total count of note-ons received (since the last reset). That number is sent out when the note-on is received, and the same number is sent out when the note is turned off.

Out 2nd outlet: Each note is also assigned a unique voice number, equal to the lowest available number. (A voice becomes available when the note assigned to it is turned off.) That number is sent out when the note-on is received, and the same number is sent out when the note is turned off.

Out 3rd outlet: The number of notes being held by **Borax** is sent out each time a note-on or a note-off is received.

Out 4th outlet: The pitch of the note-on or note-off is sent out.

Out 5th outlet: The velocity of the note-on or note-off is sent out.

Out 6th outlet: When a note-off is received,the total count of all completed notes (since the last reset) is sent out.

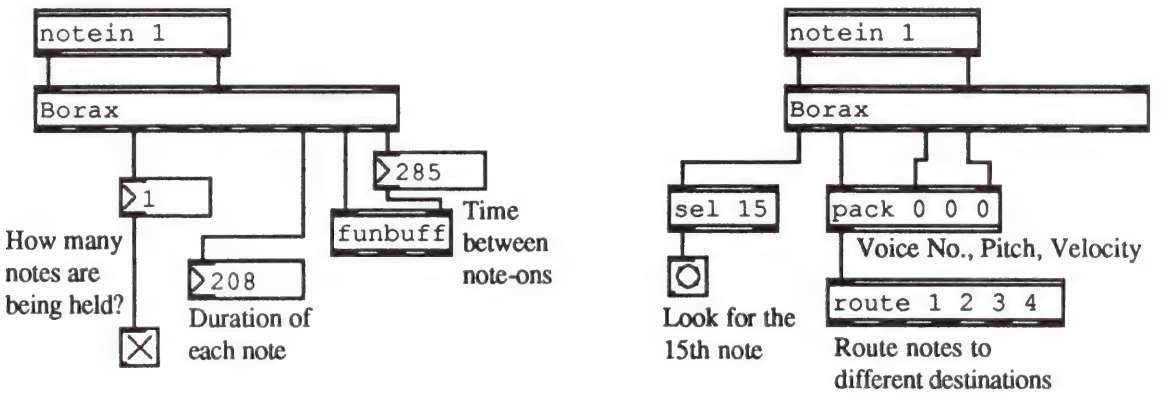
Out 7th outlet: When a note-off is received, the duration of that note, in milliseconds, is sent out.

Out 8th outlet: Each time a delta time is reported, the total count of delta times is sent out.

Out right outlet: When a note-on is received, the delta time is sent out (the time elapsed since the previous note-on, in milliseconds). A delta message in the left inlet causes the same output.

A bang received in the right inlet causes **Borax** to provide note-offs for any notes it currently holds. These note-offs trigger the same outputs as if they had actually been received.

Examples



Borax *provides extensive information about the notes passing through*

See Also

poly Allocate notes to different voices

Bucket

Pass a number from outlet to outlet, out each one in turn

Input

int — The numbers currently stored in **Bucket** are sent out, then each number is moved one outlet to the right and the new number is stored to be sent out the left outlet the next time a number is received.

float — Converted to int.

list — Only the first number in the list is used.

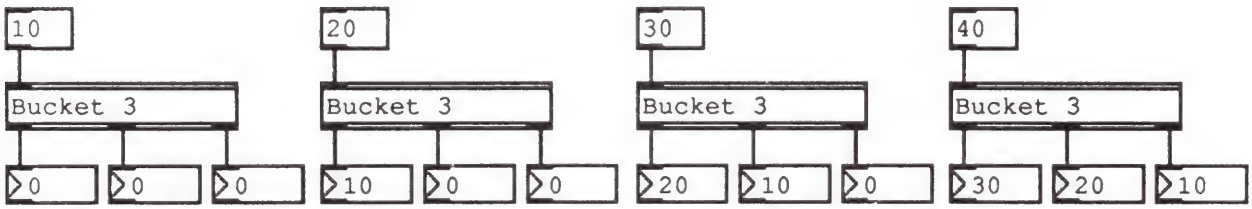
Arguments

int — Optional. Sets the number of outlets. Limited between 1 and 8. If the argument exceeds the limits, or there is no argument, there is one inlet.

Output

int — When a number is received, it is not sent out immediately, but the numbers stored in **Bucket** are sent out. The numbers are all moved one outlet to the right, and the newly received number is stored in the left position.

Examples



Stored numbers are sent out, but input is not sent out

Each time a number is received, the stored numbers are sent out, then shifted to the right to make room for the new number

Numbers are passed from one outlet to another

See Also

gate — Pass the input out a specific outlet
Decode — Send 1 or 0 out a specific outlet

Synchronize arriving numbers, output them together

buddy

Input

int — When a number has been received in *all* its inlets, **buddy** sends the received numbers out their corresponding outlets, then waits until a number has arrived again in all inlets.

A number in *any* inlet can trigger output, so long as numbers have arrived in all the other inlets since the last output.

float — Converted to int.

list — In left inlet: Each number in the list is treated as if it had been received in a separate inlet, up to the number of inlets.

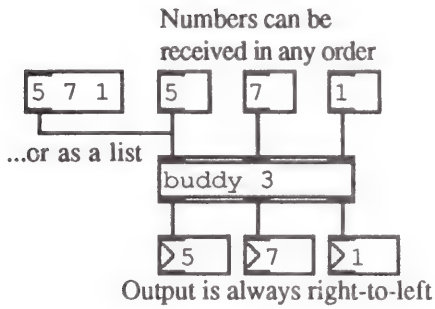
Arguments

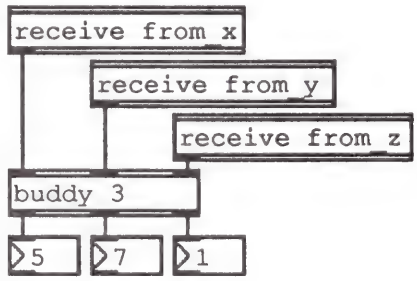
int — Optional. Sets the number of inlets (and outlets). The number of inlets is limited between 2 and 10. If there is no argument, there are two inlets.

Output

int — When a number has arrived in each inlet, the numbers are sent out the outlets, in order from right to left.

Examples





Output is synchronous, even if input is not synchronous

See Also

pack — Combine individual numbers into a list
swap — Reverse the order of two numbers
unpack — Break a list up into individual numbers

button

Flash on any message, send a bang

Input

anything — When any message is received in the inlet, **button** flashes briefly and bang is sent out the outlet. A mouse click on the **button** has the same effect.

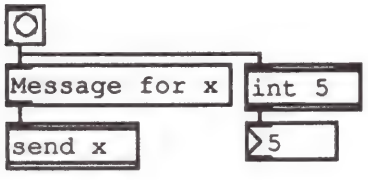
Arguments

Not applicable.

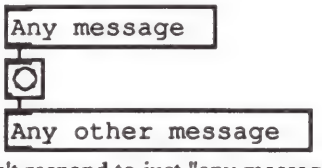
Output

bang — A mouse click or any message in the inlet causes **button** to flash and send out bang.

Examples



Triggers other messages and processes



Won't respond to just "any message"

Converts other messages to bang

See Also

bangbang — Send a bang to many places, in order
trigger — Send a number to many places, in order
ubutton — Transparent button, sends a bang
Tutorial 2 — bang means "Do it!"

Store numbers to view or edit

capture

Input

int — Numbers are stored in the order in which they are received.

list — All ints in the list are stored in order from first to last. Floats and symbols are ignored.

clear — Erases the contents of a **capture** object.

dump — Outputs the contents of the **capture** object, one number at a time, out the outlet.

Arguments

int — Optional. The first argument sets a maximum number of ints to store. If there is no argument, **capture** will store up to 512 numbers. Once the maximum has been exceeded, the earliest stored number is dropped as each new number is received.

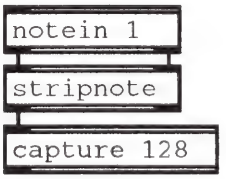
x or m — Optional. If the second argument is x, all numbers will be displayed in hexadecimal form in the editing window. If the second argument is m, numbers less than 128 are displayed in decimal, and numbers greater than 128 are in hexadecimal. If there is no argument, all numbers are displayed in decimal.

Output

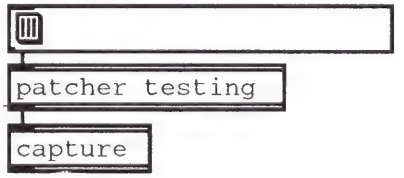
int — The captured contents are sent out the outlet, one number at a time, in response to the dump message.

Double-clicking on **capture** (when the Patcher window is locked) opens an editing window in which the stored numbers can be viewed and edited. Editing the window does not actually alter the contents of capture, but is useful for cutting and pasting values into a **table** or a separate file. (Although **capture** can continue to store numbers while the editing window is open, the editing window is not updated. It must be closed and reopened to view the newly stored numbers.)

Examples



Collect numbers to paste into a table...



...or just to see what's been going on

capture

Store numbers to view or edit

See Also

Debugging	Techniques for debugging patches
Text	Format numbers as a text file
Tutorial 34	Managing raw MIDI data

Input

bang — Reports the current location of the CD out the three leftmost outlets of the **cd** object.

drive — Followed by a number, drive switches the playing disc drive to the numbered drive in a multi-disc CD changer. Drive numbers start at 1.

eject — Ejects the currently mounted CD. eject is the same as dragging a CD icon into the trash in the Finder.

endstop — Clears any stop point set with the stop message.

ff — Begins a "fast" forward play of the CD. With no arguments, ff begins forward-playing from the current location. With one number, ff begins from the start of a track. With a list of three numbers, ff begins from a minute/second/block point.

float — Begins playing from a specified number of seconds, expressed in floating point. The value is converted into the equivalent number of minutes, seconds, and blocks by the **cd** object.

fstop — Set a stop point at a specified number of seconds, expressed in floating point. The value is converted into the equivalent number of minutes, seconds, and blocks by the **cd** object.

init — Print the table of contents for a CD in the Max window.

int — 0 pauses the CD drive, stopping the sound immediately. A number between 1 and the number of tracks begins playing the CD at the beginning of the numbered track.

list — A list of three numbers specifies an absolute minute, second, and block (1/75th of a second). The CD drive seeks to the specified location and begins playing.

pause — Pauses the CD drive. Same as 0.

resume — Resumes play from the current location if the drive is paused.

rewind — Begins a "fast" reverse play of the CD. With no arguments, rewind begins reverse-playing from the current location. With one number, rewind begins from the start of a track. With a list of three numbers, rewind begins from a minute/second/block point.

search — The word search, followed by a list of three numbers, specifies an absolute minute, second, and block. The CD drive seeks to the location but does not begin playing. Send the list of three numbers (without the word search) to begin playing at this location. Using search seems to make only a slight improvement in the time it takes to get the CD to respond after sending it a command.

stop — With no arguments, stop pauses the CD drive. Followed by a list of three numbers, stop sets an automatic stop point at a specified minute, second, and block. When the

CD reaches this location, it will automatically stop. Note that until you clear the stop point with endstop, the drive will never play past the set stop point, even if you restart the computer.

toc Send a CD's table of contents as a series of lists out the **cd** object's right outlet. The list is designed to feed directly into a **coll** object. The first number in each list is the track number, followed by the minute, second, and block at which the track starts. The last list in the series is the minute, second, and block where the last track ends. You can use the toc message to create a patch that displays the current track number based on the location of the CD.

Arguments

int The **cd** object takes two arguments, both are optional (although to specify the second argument, you must type both arguments). The first argument is the SCSI ID of the drive. If you have only one CD drive, you can type in any number from 1 to 7 for the SCSI ID and **cd** will be able to find the drive. The second argument specifies how often the **cd** object will report the time location of the drive when it is playing. A second argument of 0 means that the **cd** object will never report where the CD drive is playing unless you send it a bang message.

Output

When the **cd** object is created, it prints the current CD's *table of contents* (the minute, second, and block locations of the start of each track) in the Max window. You can use this information as a guide when you specify CD locations in minute/second/block format.

Also, as noted above, if the second argument to the **cd** object is 0 (the default), no time location output is generated. Otherwise, the numbers are sent out at an interval specified by the second argument.

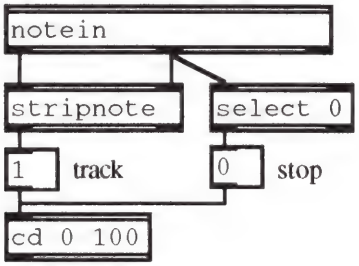
int Out left outlet: The absolute minute (starting at 1) of the CD drive when it is playing.

Out middle-left outlet: The absolute second (starting at 0) of the CD drive when it is playing.

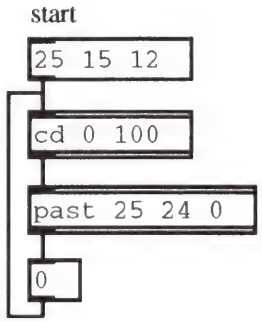
Out middle-right outlet: The absolute block number (starting at 0) of the CD drive when it is playing. A second of sound on a CD is divided into 75 blocks. Each block represents a 75th of a second, or 13 and 1/3 milliseconds.

list Out right outlet: A series of lists is output in response to the toc message. Each list has a track number followed by the minute, second, and block the track number starts at.

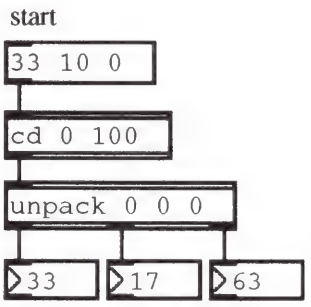
Examples



Begin playing track 1 on note-on, stop (pause CD drive) on note-off



Clicking on message box starts playing for 8 seconds and 63 blocks before turning off (triggered by the output of **past***)*



The Number boxes display the current time location of the CD drive

See Also

past Report when input increases beyond a certain value

change

Filter out repetitions of a number

Input

int — The number is sent out the outlet only if it is different from the currently stored value. Replaces the stored value.

float — Converted to int.

set — The word set, followed by a number, replaces the stored value without triggering output.

Arguments

int — Optional. Initial value for comparison to incoming numbers. If there is no argument, the initial value is 0.

Output

int — Out left outlet: The number received in the inlet is sent out only if it is different from the stored value.

Out middle outlet: If the stored value is 0 and the input is not 0, 1 is sent out; otherwise nothing is sent out.

Out right outlet: If the stored value is not 0 and the input is 0, 1 is sent out; otherwise nothing is sent out.

Examples

Filter out undesirable repetitions

See Also

TogEdge — Report a change in zero/non-zero values
!= — Compare two numbers, output 1 if not equal
Tutorial 15 — Making decisions with comparisons

Report elapsed time, at regular intervals

clocker

Input

int — In left inlet: Any non-zero number starts **clocker**. The time elapsed since **clocker** was started is sent out the outlet at regular intervals. 0 stops **clocker**.

In right inlet: The number is the time interval, in milliseconds, at which **clocker** will report the elapsed time. Any number less than 5 is set to 5. A new number in the right inlet does not take effect until the next time output is sent.

float — Converted to int.

bang — In left inlet: Starts **clocker**.

stop — In left inlet: Stops **clocker**.

ext — In left inlet: The word ext, followed by a non-zero number, synchronizes **clocker** to the MIDI Manager external clock. **clocker** reports values of the external clock accoring to "intervals" of the external clock itself, so if no timing source is connected to Max's Clock In port, or Max is not using the MIDI Manager, **clocker** will not output anything.

ext 0 sets **clocker** to use Max's internal millisecond clock.

Arguments

int — Optional. The first argument sets an initial value for the time interval at which **clocker** sends its output. If there is no argument, the initial time interval is 5 milliseconds. Any argument less than 5 will be set to 5. If the second argument is 1, **clocker** uses the MIDI Manager external clock (see the ext message discussion above). If the second argument is 0 or not present, **clocker** uses Max's internal millisecond clock.

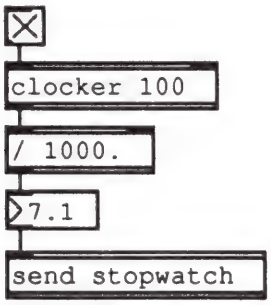
Output

int — The time elapsed, in milliseconds, since **clocker** was started. The first output is always 0, sent immediately each time **clocker** is started.

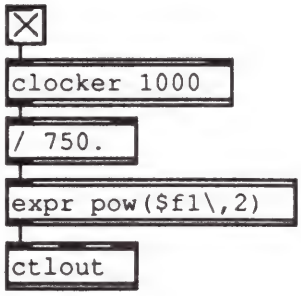
clocker

Report elapsed time, at regular intervals

Examples



Get the elapsed time



Generate numbers as a function of time

See Also

metro	Output bang, at regular intervals
tempo	Output numbers at a metronomic tempo
timer	Report elapsed time between two events
Ports	How MIDI ports are specified
Tutorial 31	Using timers

Send bang when Patcher window is closed

closebang

Input

There are no inlets. Output occurs when the Patcher window is closed.

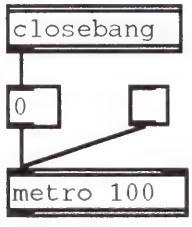
Arguments

None.

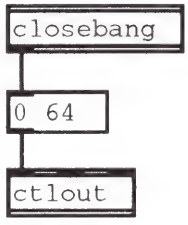
Output

bang Sent automatically when the Patcher window is closed.

Examples



Stop a process when window is about to be closed.



...or turn off sustain pedal

See Also

active	Send 1 when window is active, 0 when window is inactive
button	Flash on any message, send a bang
loadbang	Send a bang automatically when patch is loaded
Tutorial 40	Automatic actions

coll

Store and edit a collection of different messages

Input

list — The first number is used as the *address* (the storage location within **coll**) at which to store the remaining items in the list. The address will always be stored as an int.

int — The number refers to the address of a message stored in **coll**. If a message is stored at that address, the message is sent out the left outlet.

store — The word store, followed by some symbol (usually a word), followed by a message, stores the message at an address named by the symbol. (Example: store triad 0 4 7 will store the list 0 4 7 at an address named triad.)

symbol — The symbol refers to the address of a message stored in **coll**. If a message is stored at the address named by the symbol, the message is sent out the left outlet. The symbol may, but need not necessarily, be preceded by the word symbol.

float — Converted to int when received alone as an address. Left alone when part of a stored message.

goto — The word goto, followed by a number or a symbol, sets a pointer at the address specified by the number or symbol. If no such address exists, the pointer is set at the beginning of the collection. The pointer is set at the beginning of the collection initially, by default.

next — Sends the address pointed to by the pointer out the right outlet, and sends the message stored at that address out the left outlet, then sets the pointer to the next address. If the address is a symbol rather than a number, 0 is sent out the right outlet. If the pointer is currently at the last address in the collection, it *wraps around* to the first address. (Note: Number addresses are stored in ascending order. Symbol addresses are stored in the order in which they were added to the collection, after all of the number addresses.)

bang — Same effect as next.

prev — Causes the same output as the word next, but the pointer is then decremented rather than incremented. If the pointer is currently at the first address in the collection, it *wraps around* to the last address.

assoc — The word assoc, followed by a symbol and a number, *associates* the symbol with the address specified by the number, provided that the number address already exists. From then on, any reference to that symbol will be interpreted by **coll** as a reference to the number address. Each number address can have only one symbol associated with it. (Note: If the symbol was already being used as an address, or was already associated with a number address, the message that was stored at that address is removed.)

deassoc — The word deassoc, followed by a symbol and a number, removes the association between the symbol and the number address. The symbol no longer has any meaning to **coll**.

dump — Sends *all* of the stored addresses out the right outlet and *all* of the stored messages out the left outlet, in the order in which they are stored.

clear — Erases everything from the collection.

remove — The word remove, followed by a number or a symbol, removes that address and its contents from the collection.

delete — Functions similarly to the word remove, except that if the specified address is a number, all addresses of a greater number are decremented by 1.

insert — The word insert, followed by a number and a message, inserts the message at the address specified by the number, incrementing all equal or greater addresses by 1 if necessary.

length — Counts the number of messages contained in **coll** and sends the number out the left outlet. This message works well in conjunction with the **grab** object.

merge — The word merge, followed by an address and a message, appends its message at the end of the message already stored at that address. If the address does not yet exist, it is created.

max — Determines the *maximum* single numerical value (i.e. not a list or symbol) stored in the **coll** and sends the number out the left outlet. This message works well in conjunction with the **grab** object.

min — Determines the *minimum* single numerical value (i.e. not a list or symbol) stored in the **coll** and sends the number out the left outlet. This works well in conjunction with the **grab** object.

nth — The word nth, followed by an address and a number, gets the *nth* item (specified by the number) from the message at that address, and sends it out the left outlet. (Example: nth pgms 4 outputs the 4th item in the message stored at the address named pgms.)

sub — The word sub, followed by an address, an item number, and another number, replaces one item stored at the address. (Example: sub pgms 4 7 puts the number 7 in place of the 4th item of the message stored at the address pgms.) Only number values—not symbols—can be substituted in this manner.

flags — Normally, the contents of **coll** are not saved as part of the patch when the Patcher window is closed. The message flags 1 0 sets the **coll** object to save its contents as part of the Patcher that contains it. The message flags 0 0 causes the contents of the **coll** not to be saved with the Patcher that contains it.

(Get Info...) — A **coll** object can be set to save its contents as part of the patch that contains it. When the Patcher window is unlocked, select the **coll** object, choose **Get Info...** from the Max menu, and check *Save coll with Patcher*.

write — Calls up the standard **Save As...** dialog box, enabling the user to save the contents of **coll** as a separate file.

read — Calls up the standard **Open...** dialog box, enabling the user to load the contents of another file into **coll**.

refer — The word refer, followed by the name of another **coll** object, changes the **coll** receiving the message to refer to the data in the named **coll** object.

In addition to reading messages in from another file and storing messages via the inlet, one can also enter messages in **coll** by typing. Double-clicking with the mouse on the **coll** object displays the contents as text in an editing window which the user can modify.

In order to edit a collection by hand or read in from another file, it is essential to know the correct text format for the contents of a **coll** object. Each message is stored in the **coll** object on a separate line. The format of each line is as follows: the address (an int or a symbol), any symbols associated with that address (if the address is an int), a comma (to separate the address from the data it contains), the data (anything), and a semicolon to indicate the end of each line. In a line such as...

```
3 reset, set 4.7;
```

...3 is the number of the address, reset is a symbol associated with that address, and the message it contains is set 4.7. Here is how we would store the numbers 100, 200, 300, and 400 with the addresses 1, 2, 3, and 4.

```
1, 100;
2, 200;
3, 300;
4, 400;
```

Arguments

any symbol — Optional. Name of a file to be read into **coll** automatically when the patch is loaded. The information in the file must be in the correct format in order to be read in by **coll**. All **coll** objects which share the same name always share the same contents. You can use the file name as an identifier for the purpose of sharing data between multiple **coll** objects, without there needing to be an actual file with the specified name.

Store and edit a collection of different messages

coll

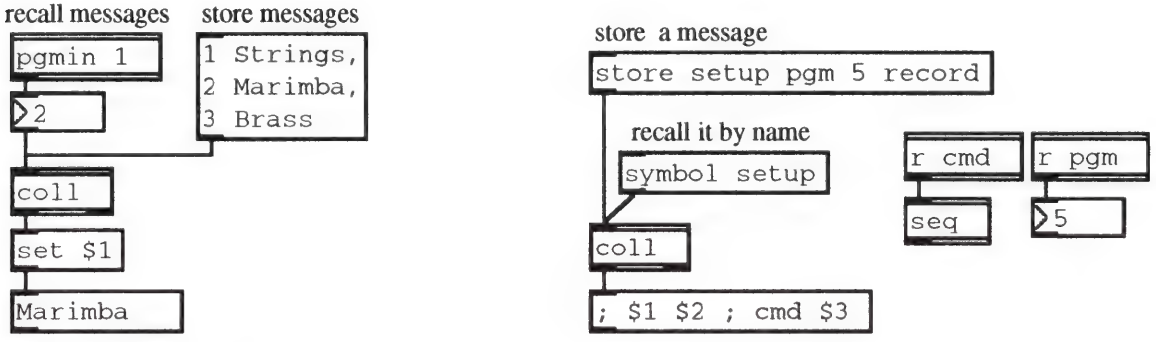
Output

anything — Messages stored in **coll** are sent out the left outlet. If the message consists of only a single symbol, it will be preceded by the word symbol when it is sent out.

int — Out left outlet: The number of messages contained in **coll** is sent out in response to the length message.

int or symbol — Out right outlet: The address is sent out whenever a message out the left outlet is triggered by bang, dump, next, or prev.

Examples



Complex messages can be recalled with a single number or word

See Also

Data Structures	Ways of storing data in Max
bag	Store a collection of numbers
table	Store and graphically edit an array of numbers
funbuff	Store x,y pairs of numbers together
Tutorial 37	Data structures

comment

Put explanatory notes or labels in a patch

Input

anything — The **comment** object has no inlets and receives no input.

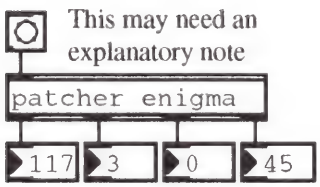
Arguments

Text is typed directly into the **comment** box when the Patcher window is in Edit mode. When the Patcher window is locked, the outline of the **comment** box disappears, and only the text is shown. The appearance of a **comment** can be modified by changing the font and by resizing its box.

Output

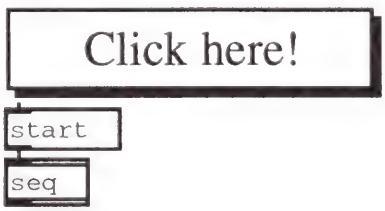
A **comment** has no outlets, sends no output, and does not affect the functioning of the patch.

Examples



Elucidate

Label



Make functional (with **ubutton***)*

See Also

ubutton — Transparent button
Tutorial 5 — toggle and comment

Count the bangs received, output the count

counter

Input

bang — In left inlet: Sends out the current count of the bangs received in the left inlet.

In left-middle inlet: Changes the direction of the count.

In middle inlet: Resets the count to its specified minimum value, which will be sent out the next time a bang is received in the left inlet.

In right-middle inlet: Resets the count to its specified minimum value, and sends out that value immediately.

In right inlet: Resets the count to its specified maximum value, which will be sent out the next time a bang is received in the left inlet.

int — In left inlet: Same effect as bang.

In left-middle inlet: Sets the direction of the count. 0 causes **counter** to count up, 1 causes it to count down, and 2 causes it to count up and down.

In middle inlet: The number sets the counter to a new value, to be sent out the next time a bang is received in the left inlet. If the number is less than the current minimum value, the minimum will be reset to that number. If the number is greater than the current maximum value, the counter will be set to that number, but the maximum value actually remains the same and the minimum is set equal to the maximum.

In middle-right inlet: The number sets the counter to a new value and sends it out immediately. If the number is less than the current minimum value, the minimum will be reset to that number. If the number is greater than the current maximum value, the number is sent out, but the maximum value actually remains the same and the minimum is set equal to the maximum.

In right inlet: Resets the maximum value sent out by **counter**. If the number is less than the current minimum, the maximum is set to be 1 greater than the minimum.

float — In left inlet: Same effect as bang.

set — In left inlet: The word set, followed by a number, sets the counter to that number, which will be sent out the next time a bang is received in the left inlet.

jam — In left inlet: The word jam, followed by a number, sets the counter to that number and sends the number out immediately.

goto — In left inlet: Same effect as set.

up — In left inlet: Causes **counter** to count up.

down — In left inlet: Causes **counter** to count down.

counter

Count the bangs received, output the count

updown — In left inlet: Causes **counter** to count in an up until it reaches the specified maximum, then count down until it reaches the specified minimum, then up, then down, and so on.

next — In left inlet: Same as bang.

min — In left inlet: The word min followed by a number, resets the minimum value of **counter** to that number. If the number is greater than the current maximum value, the minimum is set equal to the maximum.

max — In left inlet: The word max followed by a number, resets the maximum value of **counter** to that number. If the number is less than the current minimum value, the maximum is set to be 1 greater than the minimum.

Arguments

int — Optional. If there is only one argument, it sets an initial maximum count value for **counter**. If there are two arguments, the first number sets an initial minimum value, and the second number sets an initial maximum value. If there are three arguments, the first number specifies the direction of the count, the second number is the minimum, and the third number is the maximum. If there are no arguments, the direction is up, the minimum is 0, and there is no maximum.

Output

int — Out left outlet: When bang, next, or a number is received in the left inlet, the current count is sent out, within the minimum and maximum limits specified. If the direction of the count is both up and down, the count is *folded back* in the other direction when it reaches the specified limits. If the count is in only one direction, up or down, the count is *wrapped around* to the opposite extreme when it reaches its limit.

When the direction is up, or up and down, **counter**, begins counting from the specified minimum value. When the direction is down, **counter** begins from the maximum value.

Out left-middle outlet: When the count is moving downward and reaches the minimum limit, the number 1 is sent out. When the count leaves the minimum limit, 0 is sent out.

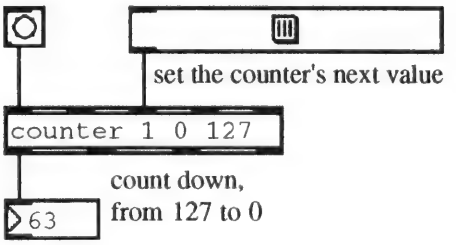
Out right-middle outlet: When the count is moving upward and reaches the maximum limit, the number 1 is sent out. When the count leaves the maximum limit, 0 is sent out.

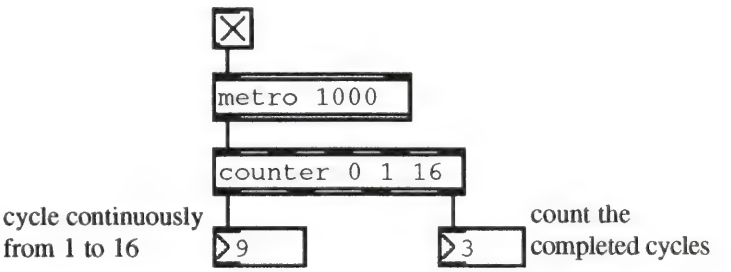
Out right outlet: An additional count is kept of the number of times **counter** reaches its maximum limit. Each time the maximum is reached, that count is sent out.

Count the bangs received, output the count

counter

Examples





Keep track of how many events have occurred, or create a continuous loop

See Also

Loops	Using loops to perform repeated operations
tempo	Output numbers at a metronomic tempo
Tutorial 31	Using timers

ctlin

Output incoming MIDI control values

Input

(MIDI) **ctlin** receives its input from a MIDI control change message received directly from the serial ports, or from the *virtual ports* of the MIDI Manager and/or OMS.

port The word port, followed by a letter a-z or the name of an OMS Input Device, sets the port from which the object receives incoming control messages. The word port is optional and may be omitted.

Arguments

a-z Optional. Specifies a single port from which to receive incoming control messages. If there is no letter present as an argument, **ctlin** can receive from all ports.

If the MIDI Manager is being used, a letter a-f specifies a virtual port. If the MIDI Manager is not being used, the letter a specifies the modem port and b specifies the printer port. If OMS is being used, the letter a-z can be an abbreviation for any OMS or MIDI Manager port (configured using the **MIDI Setup...** dialog).

OMS name Optional. When using OMS, you may use the name of an OMS Input Device as an argument to specify a port instead of a-z.

int Following the (optional) port argument, the next argument is a single controller number to be recognized by **ctlin**. If there is no controller number, or if the argument is a negative number, **ctlin** recognizes *all* controller numbers. If a single controller number *is* specified in the argument, the outlet which normally sends the controller number is unnecessary, and is not created.

Following the controller number argument is a single channel number on which to receive control messages. If the channel argument is not present, **ctlin** receives control messages on all channels. In order for this argument to be used, a controller number argument *must* precede it. To specify a channel number without specifying a controller number, use -1 for the controller number.

If a single channel number is specified as an argument, the outlet which normally sends the channel number is unnecessary, and is not created. If a port has been specified with a letter argument, channel numbers greater than 16 will be *wrapped around* to stay within the 1-16 range. If no port argument is present, a channel number can be used in place of a letter and number combination. Numbers 1-16 specify a channel number on port a, numbers 17-32 specify channels 1-16 on port b, etc.

Output

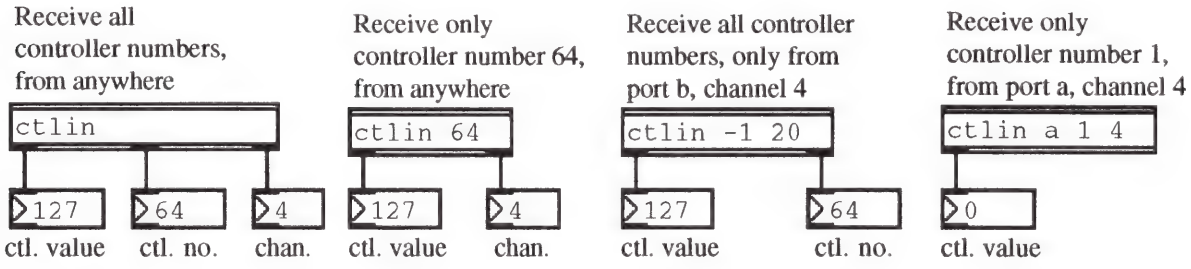
int Out left outlet: The number is the control value of an incoming MIDI control change message.

ctlin

If a specific controller number is not specified as an argument, the controller number is sent out the 2nd outlet.

If a specific channel number is not included in the argument, the channel number is sent out an additional, right, outlet.

Examples



Control messages can be filtered in a variety of ways

See Also

ctlout	Transmit MIDI control messages
MIDI	MIDI software protocol
midiin	Output incoming raw MIDI data
OMS	Using Max with OMS
Ports	How MIDI ports are specified
Tutorial 16	More MIDI ins and outs

ctlout

Transmit MIDI control messages

Input

int — In left inlet: The number is used as the control value, and **ctlout** transmits a MIDI control change message. Numbers are limited between 0 and 127.

In middle inlet: The number is stored as the controller number of the control change messages transmitted by **ctlout**. Numbers are limited between 0 and 127.

In right inlet: The number is stored as the channel number on which to transmit the control messages.

float — Converted to int.

list — In left inlet: The first number is the control value, the second the controller number, and the third the channel number. **ctlout** transmits a MIDI control change message using these values.

port — In left inlet: When not using OMS, the word port, followed by a letter a-f, specifies the port used to transmit MIDI control messages. When using OMS, port can be followed by a-z or the name of an OMS Output Device. In either case, the word port is optional and can be omitted.

Arguments

a-z — Optional. Specifies the port for transmitting MIDI control messages. If there is no argument, **ctlout** initially transmits out port a, on channel 1. When a port is specified by a letter argument, channel numbers greater than 16 received in the right inlet will be *wrapped around* to stay within the 1-16 range.

OMS name — Optional. When using OMS, the name of an OMS Output Device may be used as the first argument to specify the port.

int — Following the (optional) port argument, the next argument is an initial value for the controller number to be used in control messages transmitted by **ctlout**. Controller numbers are automatically limited between 0 and 127. If there is no controller number specified, the initial controller number is 1.

Following the controller number argument is an initial value for the channel number on which to transmit control messages. If the channel argument is not present, **ctlout** initially transmits control messages on channel 1. In order for this argument to be used, a controller number argument *must* precede it.

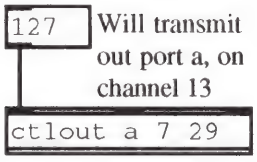
If a port has been specified with a letter argument, channel numbers greater than 16 will be *wrapped around* to stay within the 1-16 range. If no port argument is present, the channel number specifies both the port and the channel. Numbers 1-16 specify a channel number on port a, numbers 17-32 specify channels 1-16 on port b, etc.

If the MIDI Manager is being used, a letter a-f specifies a virtual port. If the MIDI Manager is not being used, the letter a specifies the modem port and b specifies the printer port. If OMS is being used, the letter a-z can be an abbreviation for any OMS or MIDI Manager port (set using the **MIDI Setup...** dialog).

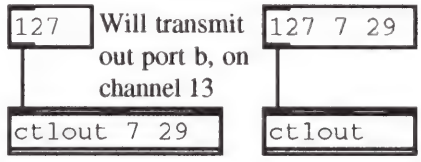
Output

(MIDI) There are no outlets. The output is a MIDI control message transmitted directly to the serial ports, or to the *virtual ports* of the MIDI Manager.

Examples



Letter argument transmits to only one port



Otherwise, number specifies both port and channel

See Also

ctlin	Ouput incoming MIDI control values
MIDI	MIDI overview and specification
midiout	Transmit raw MIDI data
OMS	Using Max with OMS
Ports	How MIDI ports are specified
Tutorial 16	More MIDI ins and outs

date

Report current date and time

Input

date — Outputs the current date as a list (month/day/year) out the left outlet.

ticks — Outputs the current value of Ticks (number of 60ths/sec since system startup) out the right outlet.

time — Outputs the current time as a list (military hours/minutes/seconds) out the middle outlet.

Arguments

None.

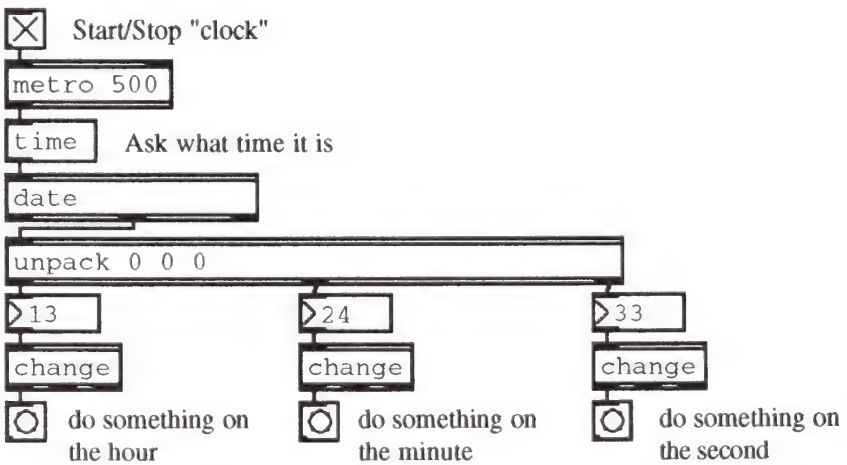
Output

list — Out left outlet: When the date message is received, **date** sends the current date as a list.

list — Out middle outlet: When the time message is received, **date** sends the current time as a list.

int — Out right outlet: When the ticks messags is received, **date** sends the current value of Ticks.

Examples



For pieces which change slowly, **date** *can be used as a clock to trigger events*

This page intentionally left blank

Decode

Send 1 or 0 out a specific outlet

Input

int **Decode** acts as a hierarchical switchboard. The right inlet is the master switch, which can turn off (send 0 out) all outlets. The middle inlet is a submaster switch, which can turn on (send 1 out) all outlets, *provided* they have not all been turned off by the master switch. The left inlet can turn on one of the outlets exclusively, provided neither the submaster switch nor the master switch is active.

In left inlet: The number specifies an outlet out to turn on, turning off all other outlets. (Whenever an outlet is turned on that was previously turned off, a 1 is sent out. Conversely, whenever an enabled outlet is disabled, a 0 is sent out.) The outlets are referred to by number, beginning with 0 on the left, and numbers received in the left inlet are automatically limited between 0 and the number of outlets minus 1.

In middle inlet: Any number other than 0 enables all disabled outlets (sends a 1 out them), unless *all* outlets are disabled. When 0 is received, **Decode** turns off all outlets except the one that had previously been on.

In right inlet: Any number other than 0 disables all enabled outlets (sends a 0 out them). Once all outlets have been disabled in this manner, no outlet can be enabled until a 0 is received in the right inlet. When a 0 is received, **Decode** re-enables all outlets that it had just disabled.

float Converted to int.

Arguments

int Optional. Sets the number of outlets. The number of outlets is limited between 1 and 8. The default is one outlet.

float Converted to int.

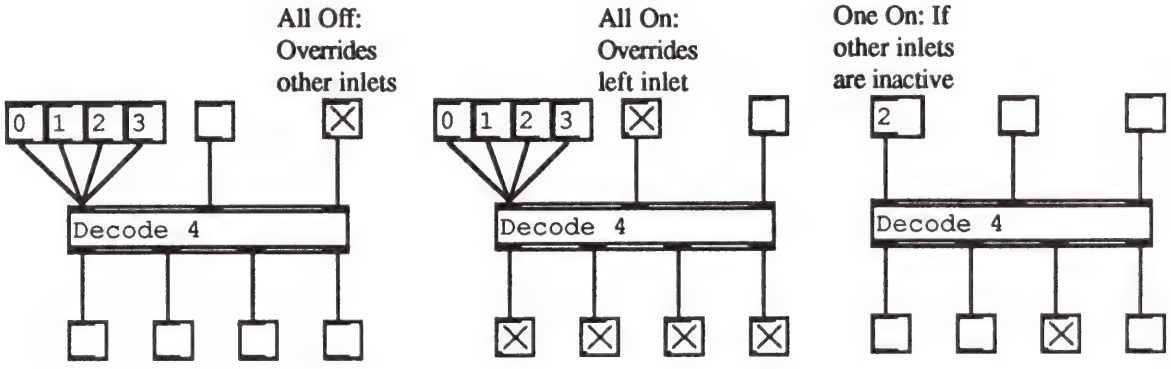
Output

int When an outlet is enabled that was previously disabled, a 1 is sent out that outlet. When an outlet is disabled that was previously enabled, a 0 is sent out that outlet. The left outlet is initially enabled.

Send 1 or 0 out a specific outlet

Decode

Examples



Decode *is a hierarchical on/off switch*

See Also

Bucket	Pass a number from outlet to outlet, out each one in turn
gate	Pass the input out a specific outlet
toggle	Switch between on and off (1 and 0)

delay / del

Delay a bang before passing it on

Input

bang — In left inlet: A bang is delayed a certain number of milliseconds before being sent out the outlet.

stop — In left inlet: Stops **delay** from outputting the bang it is currently delaying.

int — In right inlet: The number is stored as the number of milliseconds to delay a bang received in the left inlet.

float — Converted to int.

Arguments

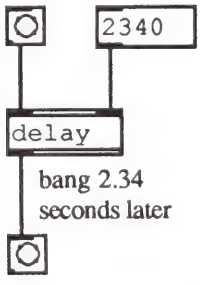
int — Sets an intial value for the number of milliseconds to delay a bang received in the left inlet. If there is no argument, the initial value is 0.

float — Converted to int.

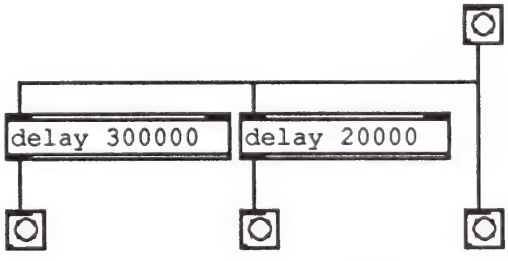
Output

bang — A bang received in the left inlet is delayed by the number of milliseconds specified by the right inlet, then is sent out the outlet. Only one bang at a time can be delayed by **delay**. If a bang is already in **delay** when a new bang is received in the left inlet, the first bang is forgotten.

Examples



Bang is delayed for a certain time



Can be used to send triggers at specific times

See Also

pipe — Delay numbers or lists
Tutorial 22 — Delay lines

Output numbers by moving a dial onscreen

dial

Input

int — The number received in the inlet is displayed graphically by **dial**, and is passed out its outlet. Optionally, **dial** can multiply the number by some amount and add an offset to it before sending it out the outlet.

The **dial** will also send out numbers in response to clicking or dragging on it directly with the mouse.

float — Converted to int.

bang — Sends out the number currently stored in **dial**.

set — The word set, followed by a number, changes the displayed value of the **dial**, without triggering output.

size — The word size, followed by a number, resets the range of the **dial**. A size of 1 will disable the **dial**, graying it out. Any number less than 1 will be set to 2.

Arguments

The range of **dial** is set by selecting it (when the window is in **Edit** mode) and choosing **Get Info...** from the Max menu. Numbers received in the inlet are automatically limited between 0 and the number 1 less than the specified range.

The **Get Info...** dialog box also provides a *Multiplier*—by which all numbers will be multiplied before being sent out, and an *Offset*—which will be added to the number, after multiplication. A newly created **dial** has a range of 128, a multiplier of 1, and an offset of 0.

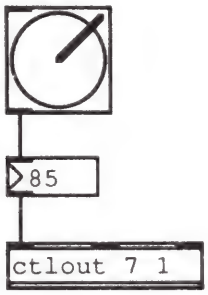
Output

int — Numbers received in the inlet, or produced by clicking or dragging on **dial** with the mouse, are first multiplied by the multiplier, then have the offset added to them, then are sent out the outlet.

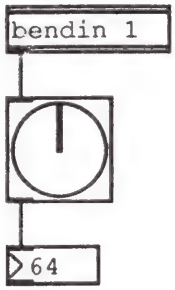
dial

Output numbers by moving a dial onscreen

Examples



Produce output by dragging onscreen...



or use to display numbers passing through

See Also

hslider, uslider	Output numbers by moving a slider onscreen
Tutorial 14	Sliders and dials

Open a dialog box for text entry

dialog

Input

symbol — The word symbol, followed by any name, opens a Macintosh dialog box prompting the user to enter text. The name argument is made the default text. Without an argument, symbol opens the dialog box using the previous default text.

bang — Opens the dialog box with the previous text displayed as the default.

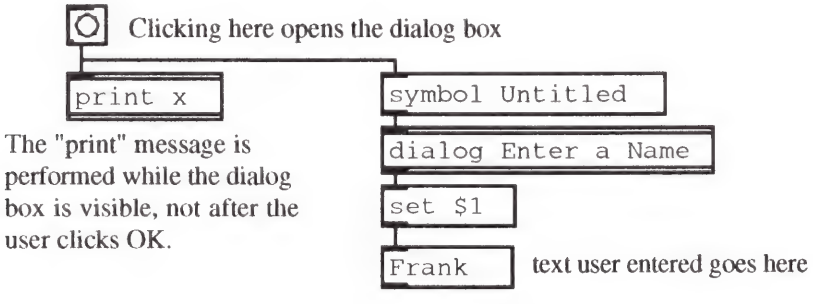
Arguments

anything — Optional. Sets the prompt which will appear above the text entry box in the dialog window.

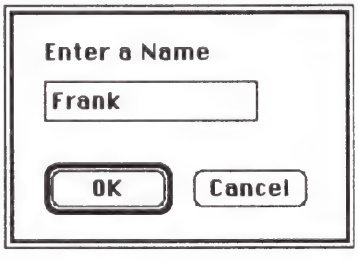
Output

symbol — If the user clicks **OK** (and not **Cancel**), **dialog** makes a symbol out of the entered text and sends it out its outlet after the word symbol. This message can be sent to a message box (as shown in the example). Because **dialog** does not "stop the clock" in order to wait for the user to type in text, you can't count on getting the typed-in symbol "immediately" after sending the message that opens the dialog box.

Examples


Clicking here opens the dialog box
print x
The "print" message is performed while the dialog box is visible, not after the user clicks OK.
symbol Untitled
dialog Enter a Name
set $1
Frank
text user entered goes here

Use a message box with set $1 *connected to another message box to display the text output from* **dialog**



The beautiful dialog box used by the **dialog** *object*

See Also

message — Send any message

This page intentionally left blank

Output random numbers in a moving range

drunk

Input

bang — In left inlet: Causes **drunk** to take a step of random size up or down from its currently stored value. It updates the stored value and sends it out the outlet.

int — In left inlet: The number replaces the stored value and is sent out the outlet.

In middle inlet: The number is stored as the maximum value that can be output by **drunk**. (Note: If the specified maximum is less than 0 it is set to 0.)

In right inlet: The number limits the step size taken in response to a bang in the left inlet. The step (up or down) will always be *less than* this number.

float — Converted to int.

list — In left inlet: The second number in the list sets the maximum value output by **drunk**, and the third number (if present) limits the step size, then the first number replaces the stored value and is sent out the outlet.

set — In left inlet: The word set, followed by a number, sets the stored value of **drunk** to that number without triggering output. The stored value is initially set in the center of the total range (1/2 the maximum value).

Arguments

int — Optional. The first argument sets an initial value for the maximum number which can be output by **drunk**. The second argument sets an inital limit on the size of random steps taken by **drunk**; the absolute value of the step size will always be *less than* this limit. If there are no typed-in arguments, the maximum value is set to 128 and the step size limit is set to 2 (movement up or down by no more than 1).

Output

int — The number sent out the outlet is automatically limited between 0 and the specified maximum value.

drunk

Output random numbers in a moving range

Examples

Numbers vary aimlessly in small steps taken within the total range

See Also

random Output a random number

Load digital signal processor code

DSP load

Input

bang Reads the code module named as the second argument (code) into the DSP node identified by the first (ID) argument.

info Prints information about the DSP node in the Max window.

Arguments

DSP load takes two arguments, a node ID and the name of the code module to load. The ID specifies the DSP node and can be specified as follows:

By vendor name:

ID	*Vendor*
MacMixExcelerator or ME	Studer Editech
AudioMedia or AM	Digidesign
SoundAccelerator or SA	Digidesign
DSP56 or D56	Ariel
Reson8 or R8	Polysonic

By slot:

If more than one board is installed in a system, they may be identified by appending _0, _1, _2, etc. to the names above. Numbering is in order of ascending NuBus slot—from left to right when viewing cards from the front of a machine. Alternatively, a NuBus slot number may be specified directly as an integer. The Reson8 is a special case. It contains 8 nodes in a single NuBus slot. Each node ID is calculated as: ((slotno+1)*100+processor_node).

By original node ID:

Use first, second, third etc to designate boards independently of vendor. First in the order are all Digidesign boards from left to right. Remaining nodes are by increasing NuBus slot ID.

The code argument names the set of files created by the DSP programmer. These files need to be in Max's search path—either in the same directory as the patch that contains the **DSP load** object, or in a subfolder called "DSP_Code_Folder."

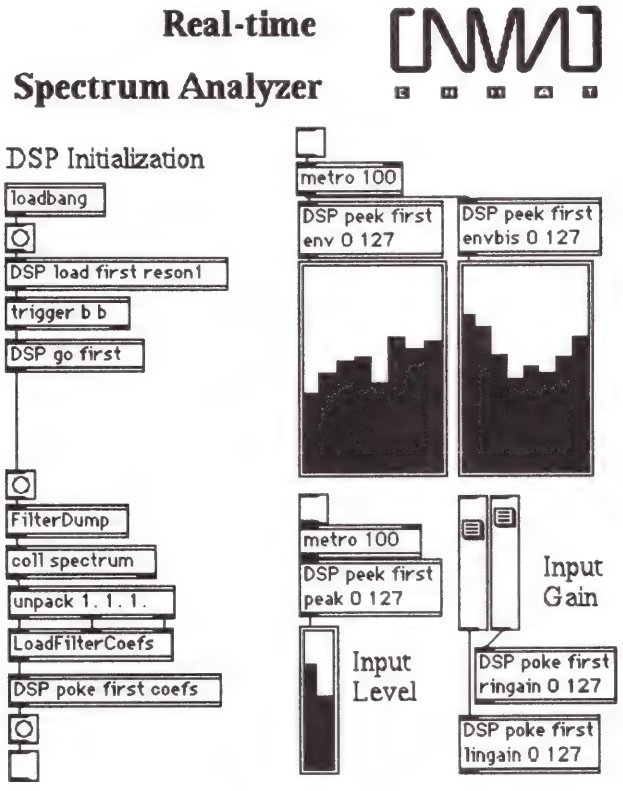
Output

bang Sent on successful completion of the load. The DSP code will be running at this point, but this will occur before a Max patcher has completed loading. This allows you to peek and poke initial values from a **loadbang** object.

DSP load

Load digital signal processor code

Examples



The four **DSP** *object variations shown in a real-time spectrum analyzer application.*

See Also

DSP go	Start digital signal processor application
DSP peek	Read values from a digital signal processor application
DSP poke	Write values to a digital signal processor application
loadbang	Send a bang automatically when patch is loaded

Start a digital signal processor application

DSP go

Input

bang Starts application previously loaded by **DSP load**.

Arguments

id The DSP node identifier (as described for **DSP load**).

Output

bang Sent after code begins running.

See Also

DSP load	Load digital signal processor code
DSP peek	Read values from a digital signal processor application
DSP poke	Write values to a digital signal processor application

DSP peek

Read values from a digital signal processor application

Input

bang — Reads a vector of data from the DSP board specified by the node ID in the object's arguments. The vector is specified by the parameter name argument to the **DSP peek** object.

info — Prints the names and descriptions of available parameters in the Max window.

offset — The word offset, followed by an integer, sets the starting offset to read data values from the vector. For an offset of n, data values are read from element n+1 of the vector.

Arguments

Arguments to **DSP peek** are the node ID (as described in **DSP load**), followed by a parameter name (which comes from a symbol table provided by the DSP code writer), followed by an optional specification for reading parameters of the form [index] from to, where index is an optional integer designating an element in the parameter vector, starting at 0. from and to specify an output range for parameters mapped from the DSP board. If from and to are specified as two integers the **DSP peek** object will output lists of integers.

Output

list — When a bang message is received in its inlet, **DSP peek** outputs a list of floats (or integers if the range arguments are integers) corresponding to the current values on the DSP board of the data vector named in the object's arguments.

See Also

DSP load	Load digital signal processor code
DSP go	Start digital signal processor application
DSP poke	Write values to a digital signal processor application
unpack	Break a list up into individual numbers

Write values to a digital signal processor application

DSP poke

Input

int — Writes a single data value to the DSP specified in the object's node ID.

float — Writes a single data value to the DSP specified in the object's node ID.

list — Writes a vector of numbers (float or integer) to the DSP specified in the object's node ID to the vector specified by the object's parameter name.

bang — Rewrites the last data vector written to the DSP.

info — Prints the names and descriptions of available parameters in the Max window.

offset — The word offset, followed by an integer, sets the starting offset to write data values to the vector. For an offset of n, data values are written to element n+1 of the vector.

Arguments

Arguments to **DSP poke** are the node ID (as described in **DSP load**), followed by a parameter name (which comes from a symbol table provided by the DSP code writer), followed by an optional specification for writing parameters of the form [index] from to, where index is an optional integer designating an element in the parameter vector, starting at 0. from and to specify how numbers received from Max are to be mapped to the DSP board.

Output

bang — After a data vector has been written successfully, **DSP poke** outputs a bang.

See Also

DSP load — Load digital signal processor code
DSP go — Start digital signal processor application
DSP peek — Read values from a digital signal processor application
pack — Combine numbers into a list

This page intentionally left blank

Evaluate a mathematical expression

expr

Input

int — The number in each inlet will be stored in place of the $i or $f argument associated with it. (Example: The number in the second inlet from the left will be stored in place of the $i2 and $f2 arguments, wherever they appear.)

float — The number in each inlet will be stored in place of the $f or $i argument associated with it. The number will be truncated by a $i argument.

symbol — The word symbol, followed by the name of a **table**, will be stored in place of the $s argument associated with that inlet, for accessing values stored in the **table**.

bang — In left inlet: Evaluates the expression using the values currently stored.

Any of the above messages in the left inlet will evaluate the expression and send out the result. A value must have been received for each changeable argument, in order for the expression to be evaluated.

The number of inlets is determined by how many changeable arguments are typed in. The maximum number of inlets is 9.

Arguments

Obligatory. The argument is a mathematical expression, in a format resembling the C programming language. The expression is made up of numbers, arithmetic operators such as + or *, comparisons such as < or >, C functions such as min() or pow(), names of **table**s, and changeable arguments ($i, $f, and $s) for ints, floats, and symbols received in the inlets.

int or float — Numbers can be used as constants in the mathematical expression.

$i or $f — A changeable int argument is specified by $i or $f and an inlet number (example: $i2). The argument will be replaced by numbers received in the specified inlet.

$s — The argument $s and an inlet number is replaced by the name of a **table** to be accessed. The argument should be immediately followed by a number in brackets specifying an address in the **table**. (Examples: $s2[7] or $s3[$i1].)

(other) — Arithmetic operators understood by **expr** are: +, -, *, /, %, ~, and ^ (to the power of). Other operators are &, &&, |, ||, and ! (not).

Many C language functions can be understood by **expr**. A function must be followed immediately by parentheses containing any arguments necessary to the function. If the function requires a comma between arguments, the comma must be preceded by a backslash (\) so that Max will not be confused by it. For example: pow($i1\,2).

expr

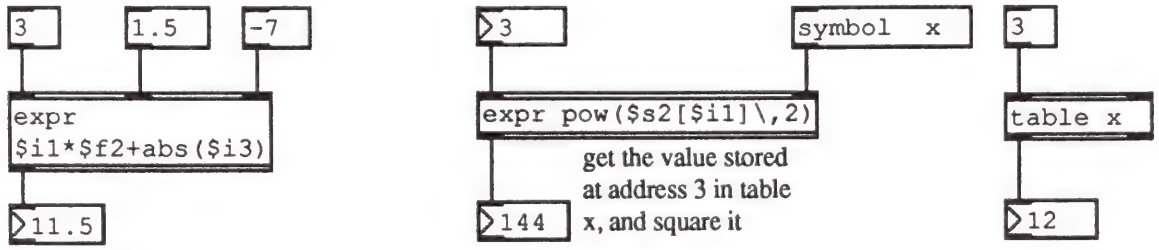
Evaluate a mathematical expression

C language functions understood by **expr** are: abs, min, max, sin, cos, tan, asin, acos, atan, atan2, sinh, cosh, tanh, int (convert to integer), float (convert to float), pow, sqrt, fact (factorial), exp (power of e to x), log10 (log), ln or log (natural log), and random. Additional functions can be added by means of external code resources placed in Max's startup folder.

Output

int or float — The output is the result of the evaluated expression.

Examples



Combine many calculations into one object, even using functions not available in other objects

See Also

if — Uses **expr** syntax for a conditional statement
Tutorial 38 — **expr** and **if**

Store a decimal number

float

Input

float — In left inlet: The number replaces the currently stored value and is sent out the outlet.

In right inlet: The number replaces the stored value without triggering output.

bang — In left inlet: Sends the stored value out the outlet.

set — In left inlet: The word set, followed by a number, replaces the stored value without triggering output.

send — In left inlet: The word send, followed by a name of a **receive** object, sends the number stored in the **float** object to all **receive** objects with that name, without sending it out **float**'s outlet.

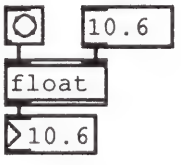
Arguments

float — Optional. Sets an initial value to be stored in **float**. If there is no argument, the initial value is 0.0. A float argument by itself, without the word float, is another way of creating and initializing a **float** object.

Output

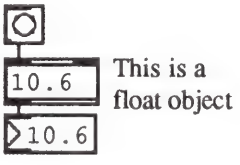
float — A number is stored in **float** as a single-precision floating point number. The precision possible in the decimal portion of the number decreases as the integer part increases. Note: Because of the way decimal numbers are stored, a float value saved in a Patcher file might be slightly altered when the file is reopened.

Examples



Output the stored value

Replace stored value and output it



Intial value is given

See Also

Data Structures	Ways of storing data in Max
int	Store an integer
Tutorial 21	Storing numbers

flush

Provide note-offs for held notes

Input

int — In left inlet: The number is treated as the pitch value of pitch-velocity pair and the note is sent out.

In right inlet: The number is stored as the velocity to be paired with numbers received in the left inlet.

list — In left inlet: The numbers must be ints. The first number is treated as the pitch, and the second number is treated as the velocity, of a pitch-velocity pair, and the numbers are sent out the outlets.

bang — In left inlet: Immediately sends note-offs for any pitches that have passed through as note-ons but not as note-offs by sending 0 out its right outlet followed by a pitch value out its left outlet.

Arguments

None.

Output

int — Out left outlet: The output is the pitch of the note-on or note-off.

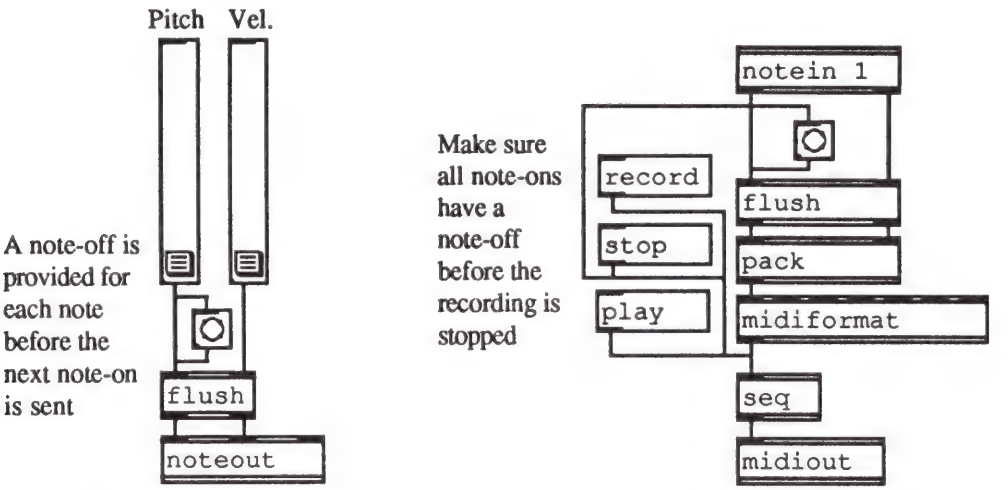
Out right outlet: The number is the velocity of the note-on or note-off.

The **flush** object keeps track of the notes that have passed through it. When a bang is received in the inlet, note-off messages are provided for any notes that have passed through as note-ons only.

Provide note-offs for held notes

flush

Examples



Make sure all notes are turned off by providing note-offs for held notes

See Also

bag	Store a collection of numbers
Borax	Report current information about note-ons and note-offs
makenote	Generate a note-off message following each note-on
stripnote	Filter out note-off messages, pass only note-on messages
sustain	Hold note-off messages, output them on command
Tutorial 13	Managing note data

follow

Compare a live performance to a recorded performance

Input

bang — Starts playing back the sequence stored in **follow**.

start — The word start by itself has the same effect as bang. The word start, followed by a number, plays the stored sequence at a tempo determined by the number. The message start 1024 indicates normal tempo. If the number is 512, **follow** plays the sequence at half the original recorded speed, start 2048 plays it back at twice the original speed, and so on.

record — Starts recording integers received in the inlet.

stop — Stops **follow** from recording, playing, or following. A stop message need not be received before switching directly from recording to playing, following to recording, etc.

append — Starts recording at the end of the stored sequence, without erasing the existing sequence.

int — When **follow** is recording, the numbers received in its inlet are recorded as a sequence. The numbers may be bytes of MIDI messages (from **midiformat** or **midiin**), exactly as with the **seq** object. However, **follow** differs from **seq** in its ability to record individual integers; with **follow** you can record notes as a single pitch value. Whether the performance is recorded as complete MIDI messages or just as note-on pitches, **follow** can effectively step through the note-on pitch numbers later, when following a performance.

When **follow** is following, numbers received in its inlet are compared to the numbers recorded in the sequence. When a number is received that matches the number in the sequence, **follow** sends out the index of that number.

float — Converted to int.

delay — The word delay, followed by a number, sets the onset time, in milliseconds, of the first event in the recorded sequence.

hook — The word hook, followed by a float, multiplies all the event times in the stored sequence by that number. For example, if the number is 2.0, all event times will be doubled, and the sequence will play back twice as slowly. Multiplications can even be performed while the sequence is playing.

write — Calls up the standard **Save As...** dialog box, so that a recorded sequence can be saved as a separate file.

read — Calls up the standard **Open...** dialog box, so that a previously recorded sequence can be read into **follow** from a separate file.

print — Prints the first few events of the recorded sequence in the Max window.

Compare a live performance to a recorded performance

follow

follow — The follow message is the main feature that distinguishes **follow** from **seq**. In effect, **follow** is like a score reader, comparing a live performance with the one previously stored.

The word follow, and a number, causes **follow** to begin comparing incoming numbers to its own stored numbers, beginning at the specified index (the specified event in its own stored sequence). When **follow** is following, and a number is received that matches the number recorded in **follow**, it sends out the index of that number.

The **follow** object is a forgiving score reader, and will try to follow along even if the incoming numbers do not exactly match the recorded sequence. If a number arrives that does not match the next number, or either of the two subsequent numbers in the sequence, **follow** does nothing. If a number arrives that matches a number up to two notes ahead in the sequence, **follow** assumes that the performer simply missed a note or two, and jumps ahead to the matched number.

next — Causes **follow** to send out the index and the stored number it is currently trying to match, and move on to the next number.

Arguments

any symbol — Optional. The argument is the name of a file containing a previously recorded sequence, to be read into **follow** automatically when the patch is loaded.

Output

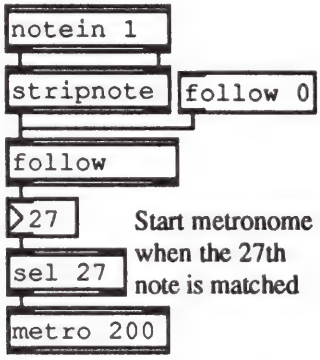
int — Out left outlet: When **follow** is following, and the number received in the inlet matches the next number in the stored sequence (or one of the two numbers after that), the index of the matched number is sent out. The index of the next number is also sent out when a next message is received.

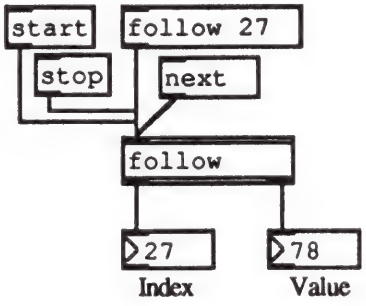
Out right outlet: When **follow** receives a bang or a start message, the recorded numbers are played back. When **follow** is following, and a next message is received, the next number in the recorded sequence is sent out.

follow

Compare a live performance to a recorded performance

Examples





A note that matches the recorded note can trigger a process, or the notes can be stepped through

See Also

seq	Sequencer for recording and playing MIDI
Sequencing	Recording and playing back MIDI performances
Tutorial 35	**seq** and **follow**

Send remote messages to a variety of objects

forward

Input

anything — Sends any message to all **receive** objects which share the name currently referred to by **forward**.

send — The word send, followed by the name of a **receive** object, sets the destination for any subsequent messages received by the **forward** object. Note that unlike a message box, forward sets its destination instantly, so you can immediately send another message to **forward** and it will be sent to the correct destination.

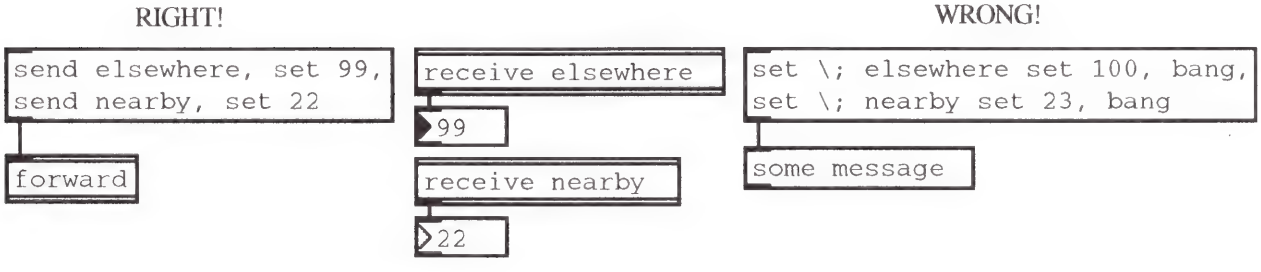
Arguments

any symbol — Optional. Sets the name for the **receive** object which will receive messages. This name can later be changed with the send message.

Output

anything — There are no outlets. A message (other than send) received in the inlet of **forward** is sent out the outlet of each **receive** with the same name, even if the **receive** is in another patch.

Examples



The right way to send messages to multiple objects at once

Because the message box does not perform the set *immediately, this scheme doesn't work*

See Also

message	Send any message
receive	Receive messages, without patch cords
send	Send messages, without patch cords
value	Share a stored number with other objects
Tutorial 24	**send** and **receive**

This page intentionally left blank

Draw framed rectangle in graphics window

frame

Input

bang — In left inlet: Draws a framed rectangle using the current screen coordinates, drawing mode, and color.

int — In left inlet: Sets the left screen coordinate of the rectangle and draws the shape.

In 2nd inlet: Sets the top screen coordinate of the rectangle.

In 3rd inlet: Sets the right screen coordinate of the rectangle.

In 4th inlet: Sets the bottom screen coordinate of the rectangle.

In 5th inlet: Sets the drawing mode of the rectangle. See the listing of drawing mode constants under **oval**.

In 6th (right) inlet: Sets the palette index (color) of the oval according to the graphics window's current palette. This setting has no effect when the monitor is in black and white mode.

priority — In left inlet: The word priority, followed by a number greater than 0, sets a **frame** object's sprite priority in its graphics window. Objects with lower priority will draw behind those with a higher priority.

Arguments

any symbol — Obligatory. The first argument to **frame** must be the name of a graphics window into which the rectangle will be drawn. The window need not exist at the time the **frame** object is created, but the rectangle will not be drawn until the name matches that of an existing and visible window.

int — Optional. Sets the initial sprite priority of the **frame**. If no priority is specified, the default is 3.

Output

(visual) — When the **frame** object's associated graphics window is visible, and a bang message or number is received in its left inlet, a shape is drawn in the window, and the object's previously drawn rectangle (if any) is erased.

Examples

See examples under **oval** or **rect**. **frame** can be directly substituted for **oval**, **rect**, or **ring**.

frame

Draw framed rectangle in graphics window

See Also

Graphics	Overview of Max graphics windows and objects
graphic	Graphics window object
oval	Draw solid oval in graphics window
rect	Draw solid rectangle in graphics window
ring	Draw framed oval in graphics window

Reverse the sequential order of two decimal numbers

fswap

Input

float — In left inlet: The number is sent out the right outlet, then the number in the right inlet is sent out the left outlet.

In right inlet: The number is stored to be sent out the left outlet when a number is received in the left inlet.

int — If there is a float argument, the numbers are converted to float. If there is an int argument or no argument, the number received in the right inlet is stored as an int.

list — In left inlet: The numbers are stored in **fswap**. The first number is sent out the right outlet, then the second number is sent out the left outlet.

bang — In left inlet: Swaps and sends out the numbers currently stored in **fswap**.

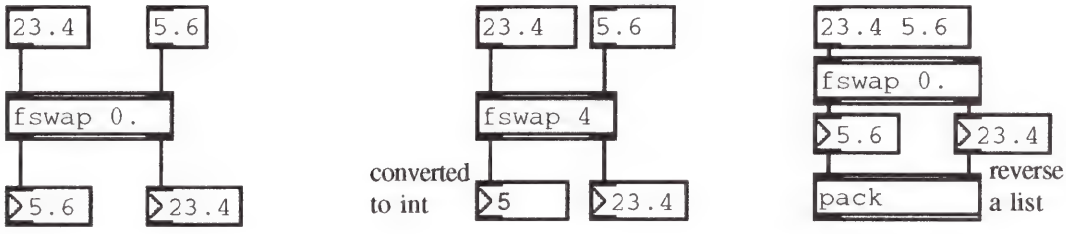
Arguments

int or float — Optional. Sets an initial value for the number which is to be sent out the left outlet. If there is no argument, the initial value is 0. If there is an int argument or no argument, an int is sent out the left outlet. (The number sent out the right outlet is *always* a float.)

Output

int — When a number is received in the left inlet, the number in each inlet is sent out the opposite outlet.

Examples



Numbers are sent out in reverse order from the order in which they were received

See Also

pack — Combine numbers together into a list
swap — Reverse the sequential order of two numbers
unpack — Break a list up into individual numbers

funbuff

Store x,y pairs of numbers together

Input

list — In left inlet: The first number is the x value, and the second number is the y value, of an x,y pair stored in **funbuff**. If the x value is the same as an x value already stored in **funbuff**, the previously stored pair is replaced by the new pair.

int — In left inlet: The number is the x value of an x,y pair. If a y value has been received in the right inlet, the two numbers are stored together in **funbuff**. Otherwise, the x value causes the corresponding y value stored in **funbuff** to be sent out the left outlet.

If there is no stored x value which matches the number received, **funbuff** uses the closest x value which is *less than* the number received, and sends out the corresponding y value.

In right inlet: The number is a y value which will be paired with the next x value received in the left inlet, and stored in **funbuff**.

clear — Erases the contents of funbuff.

goto — The word goto, followed by a number, sets a pointer to the x value (index) specified by the number. A subsequent next message will return the y value at the specified x.

min — Sends the minimum y value currently stored in the **funbuff** out the left outlet.

max — Sends the maximum y value currently stored in the **funbuff** out the left outlet.

next — Finds the x value pointed to by the pointer (or, if the pointer points to a number not yet stored as an x value, to the next *greater* x value), and sends the corresponding y value out the left outlet. Also, **funbuff** calculates the difference between that x value and the value previously pointed to by the pointer, sends the difference out the middle outlet, and resets the goto pointer to the next greater x value.

write — Calls up the standard **Save As...** dialog box, so that the contents of **funbuff** can be saved as a separate file.

read — Calls up the standard **Open...** dialog box so that a file of x,y values can be read into **funbuff**.

Arguments

any symbol — Optional. The argument specifies the name of a file to be read into **funbuff** when the patch is loaded. Changes to the contents of one **funbuff** will not affect the contents of another **funbuff** object with the same name.

A file for **funbuff** can also be created using a text editor window, beginning the text with the word funbuff, followed by a list of space-separated numbers that specify alternating x and y values. A **funbuff** which has been saved as a file can be viewed

and edited as text by choosing **Open as Text...** from the File menu. Numbers in the form of text can be pasted in from other sources such as the editing window of a **capture** object, or even from another program such as a word processor.

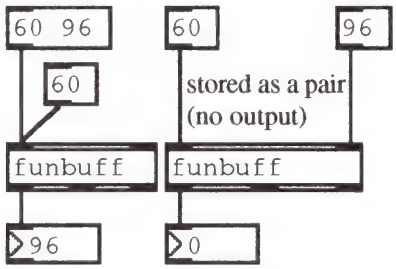
Output

int — Out left outlet: When an x value is received in the left inlet, the corresponding y value is sent out. (Or, if there is no such x value yet stored in **funbuff**, the y value corresponding to the next *lesser* x value is sent out.) When the word next is received in the left inlet, **funbuff** sends out the y value that corresponds to the x value pointed to by its pointer (or, if there is no such x value, the y value of the next *greater* x value).

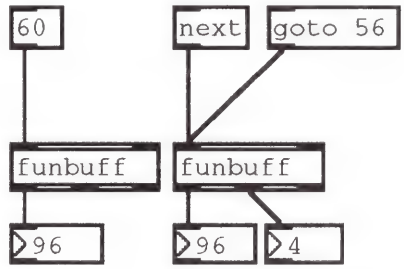
Out middle outlet: When the word next is received in its left inlet, **funbuff** sends out the difference between the x value pointed to by its pointer, and the x value *previously* pointed to, then resets the pointer to the next x value.

bang — Out right outlet: When the pointer reaches the end of a **funbuff**, no numbers are sent out in response to a next message, but a bang is sent out to notify that the end has been reached.

Examples



Lists are stored as x,y pairs



An x value alone, or next, *sends out a y value*

See Also

table — Store and graphically edit an array of numbers
Tutorial 27 — Your object

gate

Pass the input out a specific outlet

Input

int — In left inlet: The number specifies an open outlet through which to pass all messages received in the right inlet. A number in the left inlet does not trigger any output itself.

float — In left inlet: Converted to int.

bang — In left inlet: Reports the current open outlet, or 0 if closed, out the left outlet. This message is designed to be used in conjunction with the **grab** object.

anything — In right inlet: All messages are passed out the open outlet, specified by the number in the left inlet.

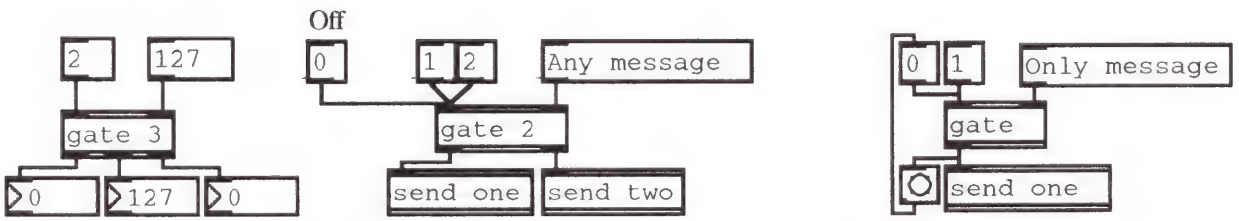
Arguments

int — Optional. Specifies the number of outlets. Limited between 1 and 10. If there is no argument, there is only one outlet.

Output

anything — Messages received in the right inlet are passed out the outlet specified by the number in the left inlet. If the number in the left inlet is 0, or if no outlet number has been received yet, all messages are ignored. If the number in the left inlet is less than 0, messages are sent out the *leftmost* outlet. If it is greater than the number of existing outlets, messages are sent out the *rightmost* outlet.

Examples



Message is passed out the specified outlet

This one closes the door behind itself

See Also

Ggate — Pass the input out one of two outlets
switch — Receive the input in a specific inlet
Tutorial 17 — Gates and switches

Pass the input out one of two outlets

Ggate

Input

int — In left inlet: The number specifies which one of the two outlets is to be open. 0 specifies the left outlet, any number other than 0 specifies the right outlet. The arrow on **Ggate** points to the open outlet.

bang — In left inlet: Causes the arrow to point to the other outlet. Clicking on **Ggate** with the mouse has the same effect.

float — In left inlet: Converted to int.

anything — In right inlet: All messages are passed out the open outlet.

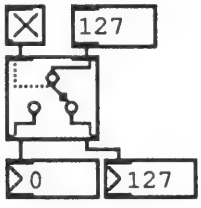
Arguments

Not applicable.

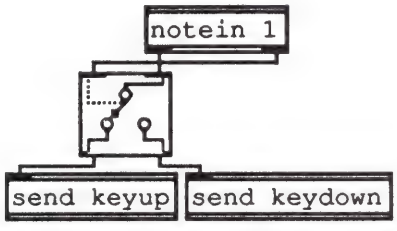
Output

anything — Messages received in the right inlet are passed out one of the two outlets. If the number in the left inlet is 0, incoming messages are sent out the left outlet. If the number in the left inlet is not 0, messages are sent out the right outlet.

Examples



Specify one of two outlets



Any comparison can be used as a criterion

See Also

gate — Pass the input out a specific outlet
Gswitch — Receive the input in one of two inlets
Tutorial 17 — Gates and switches

grab

Intercept the output of another object

Input

anything Message received in the inlet is sent out the right outlet.

Arguments

int Optional. Sets the number of outlets, *in addition to* the right outlet. If there is no argument, **grab** has 1 additional outlet.

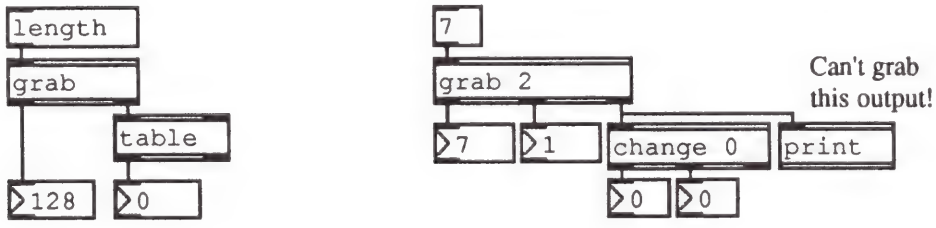
Output

anything Out right outlet: The right outlet should be connected only to *the leftmost inlet* of other objects. The message received in the inlet is sent out to the left inlet of all objects connected to the right outlet. Whatever goes out *their* outlets, however, is then *intercepted* by **grab**.

Out other outlets: Whatever would normally be sent out the outlets of the objects connected to the right outlet, is sent out **grab**'s outlets instead, in response to a message from **grab**. Whatever would be sent out the leftmost outlet of the other objects is sent out the leftmost outlet of **grab**, and so on. Note: Only the output that is sent out the *outlets* of other objects can be intercepted by **grab**. Other types of output, such as transmission of MIDI messages or printing in the Max window, cannot be intercepted by **grab**. Also, **grab** does not intercept the output of timing objects such as **seq**, **metro**, and **clocker**.

The right outlet of **grab** can also be connected to a **send** object, thus communicating with remote objects connected to a **receive** with the same name. Connecting the right outlet of **grab** to the inlet of a **patcher** object, however, will *not* grab the output of the subpatch. It will simply grab the output of the **inlet** object *inside* the subpatch, which is exactly the same as its input.

Examples



Get an object's output by "grabbing" it before it comes out the outlet

See Also

preset Store and recall the settings of other objects
table Store and graphically edit an array of numbers

Graphics
window object

graphic

Input

open — Causes the window associated with the **graphic** object to become visible. The window is also brought to the front.

close — Causes the window associated with the **graphic** object to become invisible.

Arguments

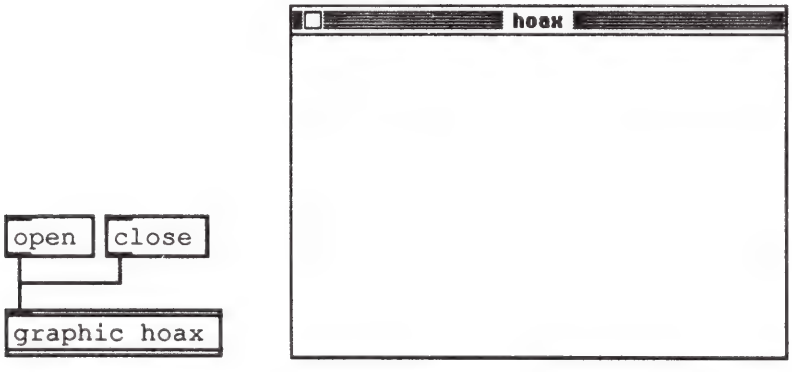
symbol — Obligatory. Identifies the **graphic** object's window. Drawing and animation objects use this symbol to draw in the window.

int — Optional. Following the name of the **graphic** object, four coordinates can be specified for the location of the window on the screen. Note that if you save a patch containing a **graphic** object, the current window location is saved when you don't type these arguments for window position. The arguments are useful in the case where you want the object's window to appear in a certain position each time the patch is opened.

Output

None. Other objects draw into a **graphic** object's window.

Examples



graphic *object that creates a window of the default size. The window can be resized by clicking in the lower right corner where you'd expect the grow box to be.*

See Also

Graphics	Overview of graphics windows and objects
oval	Draw solid oval in graphics window
pics	Animation in graphics window

Gswitch

Receive the input in one of two inlets

Input

int — In left inlet: The number specifies which one of the other two inlets is to be open. 0 specifies the middle inlet, any number other than 0 specifies the right inlet. The arrow on **Gswitch** points to the open inlet.

bang — Causes the arrow to point to the other inlet. Clicking on **Gswitch** with the mouse has the same effect.

float — In left inlet: Converted to int.

anything — In middle or right inlet: Messages received in the open inlet are passed out the outlet, while messages received in the other inlet are ignored.

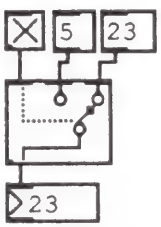
Arguments

Not applicable.

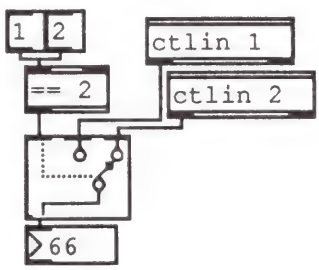
Output

anything — If the number in the left inlet is 0, all messages received in the middle inlet are passed out the outlet, and messages received in the right inlet are ignored. If the number in the left inlet is *not* 0, messages received in the middle inlet are ignored, and all messages received in the right inlet are passed out the outlet.

Examples



Specify one of two inlets



Any comparison can be used as a criterion

See Also

Ggate — Pass the input out one of two outlets
switch — Receive the input in a specific inlet
Tutorial 17 — Gates and switches

Make a histogram of the numbers received

Histo

Input

int — In left inlet: **Histo** keeps count of how many times it has received a number between 0 and 127 in the left inlet. When a number is received, **Histo** includes it in the count, sends the number of times that number has been received out the right outlet, and passes the number itself out the left outlet. Numbers outside the range 0-127 are ignored.

In right inlet: Has the same effect as a number in the left inlet, except that the number is *not* counted by **Histo**.

clear — Erases the memory of **Histo**, to begin a new histogram.

Arguments

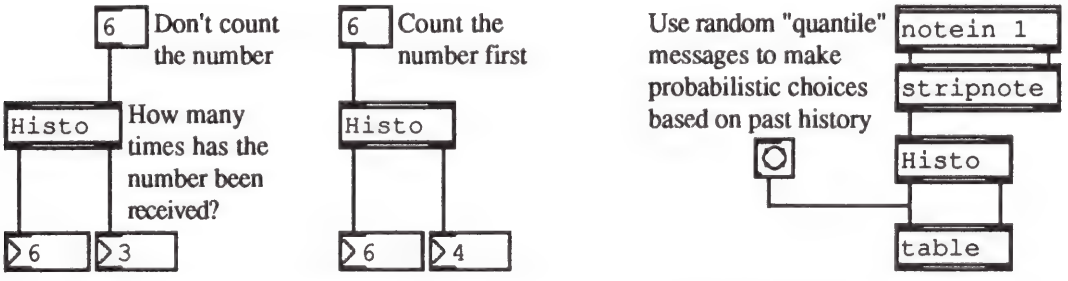
None.

Output

int — Out left outlet: The number received in the inlet.

Out right outlet: The count of the number of times that number has been received.

Examples



Store a histogram of the numbers received, or send the values to a **table**

See Also

Quantile	Using **table** for probability distribution
table	Store and graphically edit an array of numbers
Tutorial 33	Probability tables

hslider

Output numbers by moving a slider onscreen

Input

int — The number received in the inlet is displayed graphically by **hslider**, and is passed out the outlet. Optionally, **hslider** can multiply the number by some amount and add an offset to it, before sending the number out its outlet.

The **hslider** will also send out numbers in response to mouse clicking or dragging.

float — Converted to int.

bang — Sends out the number currently stored in **hslider**.

set — The word set, followed by a number, resets the value displayed by **hslider**, without triggering output.

size — The word size, followed by a number, sets the range of the **hslider**. A size of 1 will disable the hslider, graying it out. Any number less than 1 will be set to 2.

Arguments

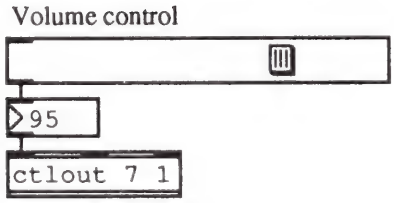
The range of **hslider** is set by selecting it (when the Patcher window is unlocked) and choosing **Get Info...** from the Max menu. Numbers received in the inlet are automatically limited between 0 and the number 1 less than the specified range.

The **Get Info...** dialog box also provides a *Multiplier*—by which all numbers will be multiplied before being sent out—and an *Offset*—which will be added to the number, after multiplication. A newly created **hslider** has a range of 128, a multiplier of 1, and an offset of 0.

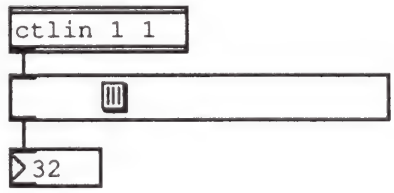
Output

int — Numbers received in the inlet, or produced by clicking or dragging on **hslider** with the mouse, are first multiplied by the *multiplier*, then have the *offset* added to them, then are sent out the outlet.

Examples



Produce output by dragging onscreen...



or use to display numbers passing through

Output numbers by moving a slider onscreen

hslider

See Also

slider, uslider	Output numbers by moving a slider onscreen
Tutorial 14	Sliders and dials

if

Conditional statement in if/then/else form

Input

int — The number in each inlet will be stored in place of the $i or $f argument associated with it. (Example: The number in the second inlet from the left will be stored in place of the $i2 and $f2 arguments, wherever they appear.)

float — The number in each inlet will be stored in place of the $f or $i argument associated with it. The number will be truncated by a $i argument.

symbol — In left inlet: The word symbol, followed by a symbol (usually a word), will be stored in place of the $s1 argument.

bang — In left inlet: Evaluates the conditional statement using the values currently stored.

Any of the above messages in the left inlet will evaluate the conditional statement and send out the result. A value must have been received for each changeable argument in order for the statement to be evaluated.

The number of inlets is determined by how many changeable arguments are typed in. The maximum number of inlets is 9.

Arguments

Obligatory. **if**'s arguments start with a *conditional statement* that uses the same syntax as **expr**. Refer to the description of the **expr** object for details. Following the **expr**-style conditional statement is the word then, followed by a message expression described below. After the message expression, there is an optional else and a second message expression.

if evaluates the conditional expression, and if the result is non-zero, evaluates the message expression after the word then. Otherwise, it evaluates the second message expression after the word else (or does nothing in the case where no else and second message expression have been typed in.

then, else — Message expressions are similar to what you type into a message box, with the following differences:

$i1,$f1,$s1 — You use $i1, $f1, or $s1 instead of $1 for changeable arguments.

send — No commas or semicolons are allowed. Messages can be sent to remote **receive** objects by preceding the message expression with send, followed by the name of the **receive** object.

out2 — The keyword out2 in a message expression creates a second, right outlet for the **if** object. If out2 precedes a message expression, the result of the expression is sent out the right outlet instead of the left outlet.

Conditional statement in if/then/else form

if

Output

anything — Messages are sent out the outlet. If the word out2 is present as an argument, there will be two outlets, and messages following out2 will be sent out the right outlet. If the word send is present as an argument, the word that follows it is the name of a **receive** object, and the message that follows it will be sent to the **receive** object.

Examples

Complex comparisons and results can be described in a single object

See Also

expr — Evaluate a mathematical expression
Tutorial 38 — **expr** and **if**

IncDec

Increment/decrement the value of a user interface object

Input

int A number sent to **IncDec**'s inlet sets the value which will be incremented or decremented by clicking on the top or bottom of half of the object. The number is not sent to the outlet. **IncDec** is designed to be used with user interface objects such as the **Number box** and various sliders and dials, as shown in the examples.

Arguments

None.

Output

int When you click on the top half of an **IncDec** object, it sends out a value which is one greater than the last value received at its inlet or sent out its outlet, whichever happened most recently. Holding the mouse button down continues to increment the output, gradually increasing in rate of output.

The same is true for the bottom half of **IncDec**, except that the values are *decremented.*

Examples

Standard arrangement with Number box

You can also arrange the patch cords in an X (and then hide them).

Use of **IncDec** *with* **Number box** *and* **hslider**

See Also

Number box	Display and output a number
hslider	Output numbers by moving a slider onscreen
uslider	Output numbers by moving a slider onscreen

This page intentionally left blank

inlet

Receive messages from outside a Patcher

Input

anything — Each **inlet** object in a patch will show up as an inlet at the top of an object box when the patch is used inside another patch (as an object or a subpatch). Messages sent into such an inlet will be received by the **inlet** object in the subpatch.

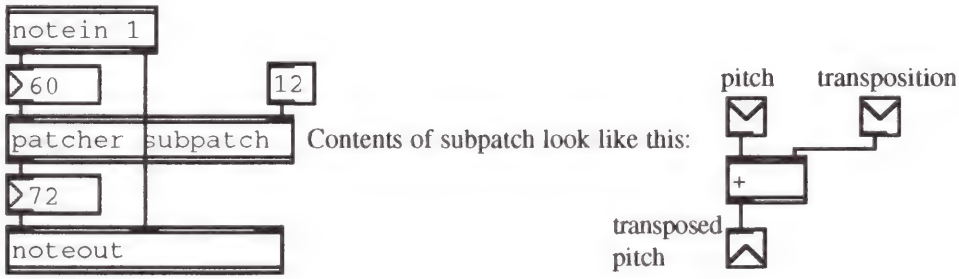
Arguments

A descriptive **Assistance** message can be assigned to an inlet by selecting the **inlet** object (when the Patcher window is unlocked) and choosing **Get Info...** from the Max menu. The **Assistance** message will appear when the patch is used as an object inside another patch.

Output

anything — In a subpatch **inlet** sends out whatever messages it receives through patch cords from the patch that contains it.

Examples



Inlets of the subpatch... *correspond to the* **inlet** *objects in the subpatch*

See Also

outlet	Send messages out from a Patcher
receive	Receive messages from other objects without patch cords
send	Send messages to other objects without patch cords
Tutorial 26	The **patcher** object

Store an integer value

int

Input

int — In left inlet: The number replaces the currently stored value and is sent out the outlet.

In right inlet: The number replaces the stored value without triggering output.

float — Converted to int.

bang — In left inlet: Sends the stored value out the outlet.

set — In left inlet: The word set, followed by a number, replaces the stored value without triggering output.

send — In left inlet: The word send, followed by the name of a **receive** object, sends the value stored in **int** to all **receive** objects with that name, without sending it out the outlet of the **int**.

Arguments

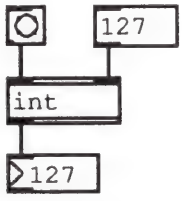
int — Optional. Sets an initial value to be stored in **int**. If there is no argument, the initial value is 0. An int argument by itself, without the word int, is another way of creating and initializing an **int** object.

float — Converted to int.

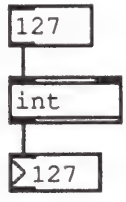
Output

int — A number is stored in **int** as a long (32-bit) integer.

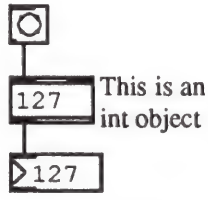
Examples



Output the stored value



Replace the stored value and output it



Initial value is given

See Also

float — Store a decimal number
receive — Receive any message remotely
Tutorial 21 — Storing numbers

iter

Break a list up into a series of numbers

Input

list	The numbers in the list are sent out the outlet in sequential order.
int	The number is sent out the outlet.
float	Converted to int.
bang	Sends the number or list most recently received, in sequential order.

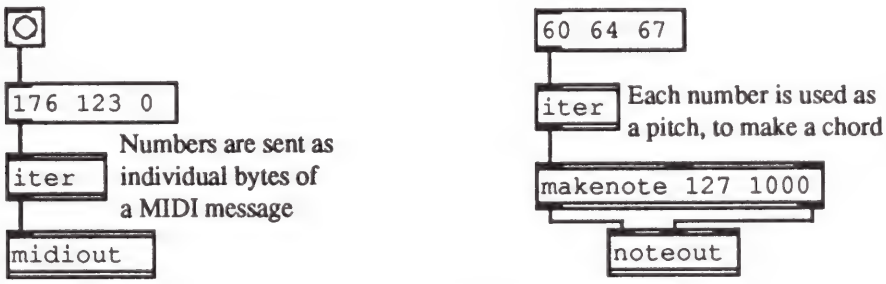
Arguments

None.

Output

int	The numbers received in the inlet are sent out one at a time.

Examples



Numbers in a list pass through **iter** *one at a time*

See Also

thresh	Combine numbers into a list, when received close together
unpack	Break a list up into individual numbers
Tutorial 30	Number groups

Report keys typed on the Macintosh keyboard

key

Input

(keyboard) The input to **key** comes directly from the Macintosh keyboard. There are no inlets.

Arguments

None.

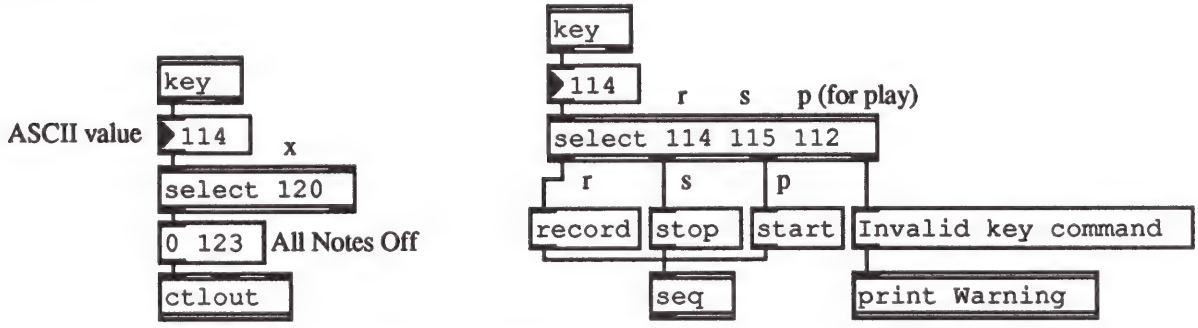
Output

int Output is sent each time a key is depressed on the Macintosh keyboard. (Holding the key down does not produce repeated output.)

Out left outlet: The ASCII value of the typed key.

Out right outlet: The key code of the typed key.

Examples



Keys typed on the Macintosh keyboard can be used to trigger messages

See Also

keyup	Report keys released on the Macintosh keyboard
numkey	Interpret numbers typed on the Macintosh keyboard
Tutorial 20	Using the Macintosh keyboard

keyup

Report keys released on the Macintosh keyboard

Input

(keyboard) The input to **keyup** comes directly from the Macintosh keyboard. There are no inlets.

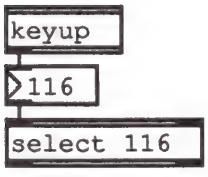
Arguments

None.

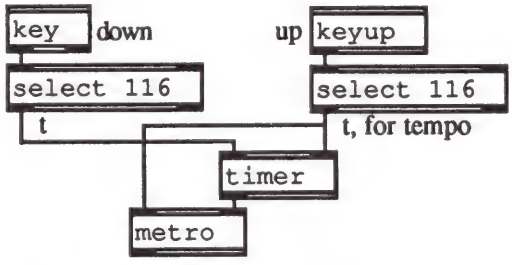
Output

int Each time a key is released on the Macintosh keyboard, its ASCII value is sent out the outlet. (Nothing is sent when the key is first depressed.)

Examples



ASCII value is sent when key is released



Used with **key** *to measure how long a key is down*

See Also

key	Report keys typed on the Macintosh keyboard
numkey	Interpret numbers typed on the Macintosh keyboard
Tutorial 20	Using the Macintosh Keyboard

Report keys typed on the Macintosh keyboard

key

Input

(keyboard) The input to **key** comes directly from the Macintosh keyboard. There are no inlets.

Arguments

None.

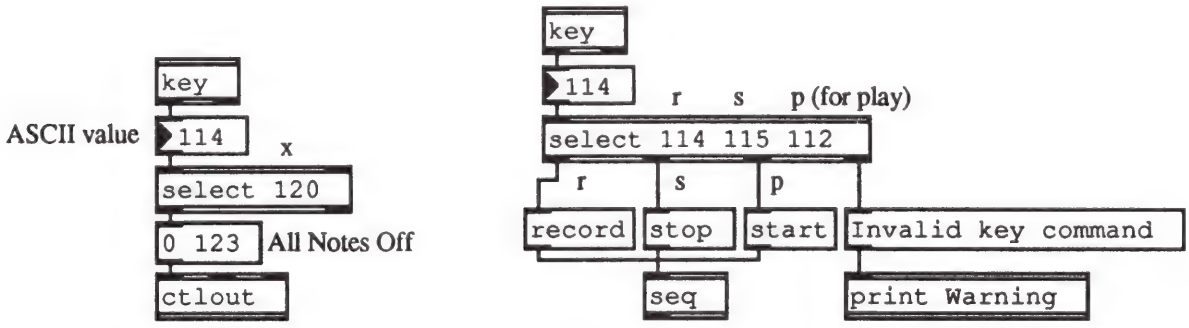
Output

int Output is sent each time a key is depressed on the Macintosh keyboard. (Holding the key down does not produce repeated output.)

Out left outlet: The ASCII value of the typed key.

Out right outlet: The key code of the typed key.

Examples



Keys typed on the Macintosh keyboard can be used to trigger messages

See Also

keyup	Report keys released on the Macintosh keyboard
numkey	Interpret numbers typed on the Macintosh keyboard
Tutorial 20	Using the Macintosh keyboard

keyup

Report keys released on the Macintosh keyboard

Input

(keyboard) The input to **keyup** comes directly from the Macintosh keyboard. There are no inlets.

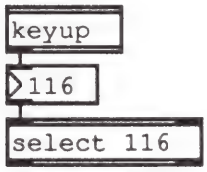
Arguments

None.

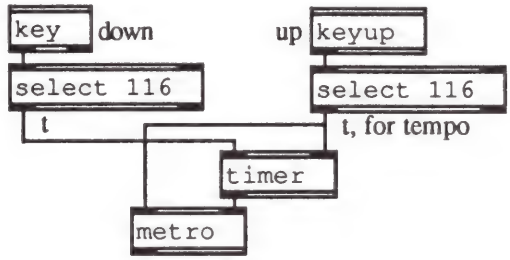
Output

int Each time a key is released on the Macintosh keyboard, its ASCII value is sent out the outlet. (Nothing is sent when the key is first depressed.)

Examples



ASCII value is sent when key is released



Used with **key** *to measure how long a key is down*

See Also

key	Report keys typed on the Macintosh keyboard
numkey	Interpret numbers typed on the Macintosh keyboard
Tutorial 20	Using the Macintosh Keyboard

Output numbers from a keyboard onscreen

kslider

Input

int — The number received in the inlet is displayed graphically by **kslider** if it falls within its displayed range. The current velocity value (from 1 to 127) that kslider holds is sent out its right outlet, followed by the received number out the left outlet.

kslider also sends out numbers when you click or drag on it with the mouse. The velocity value is determined by the vertical position of the mouse within each key. Higher vertical positions produce higher velocities, to a maximum of 127.

bang — Sends out the pitch and velocity values currently stored in **kslider**.

set — The word set, followed by a number, changes the value displayed by **kslider**, without triggering output.

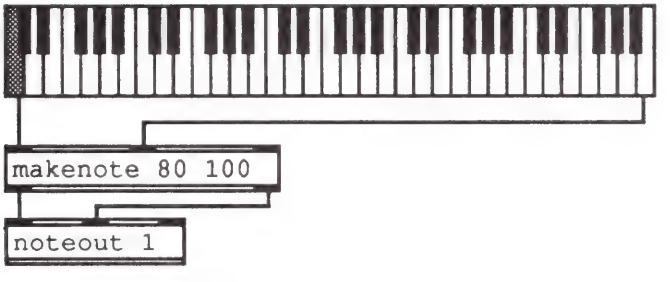
Arguments

kslider is configured by selecting it and choosing **Get Info...** from the Max menu. The Offset sets the lowest C which will be displayed, ranging from -2 (note 0) to 7. The Range can be set between 1 and 9 octaves. And there are two keyboard sizes to choose from, large and small. Both sizes are used in the examples shown below. A newly created **kslider** has a range of 5 octaves, starting at C1 (MIDI note 36).

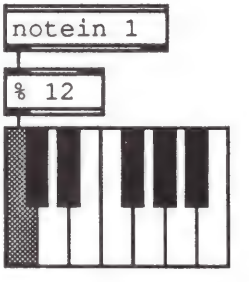
Output

int — kslider sends its current velocity value out its right inlet, followed by the (displayable) pitch value out its left outlet, when a number is received in its inlet or your click or drag on the object.

Examples



Produce output by clicking on the keyboard...



or use to display incoming pitches

See Also

slider, uslider — Output numbers by moving a slider onscreen
Tutorial 14 — Sliders and dials

led

Display on/off status in color

Input

int — If the number is 0, **led** shows its darkened state, and outputs 0. If the number is not 0, **led** shows its brightened state and outputs 1.

float — Converted to int.

bang — Flashes **led** on and off quickly, and outputs 0.

Clicking on an **led** toggles it back and forth between bright and dark, outputting 1 and 0.

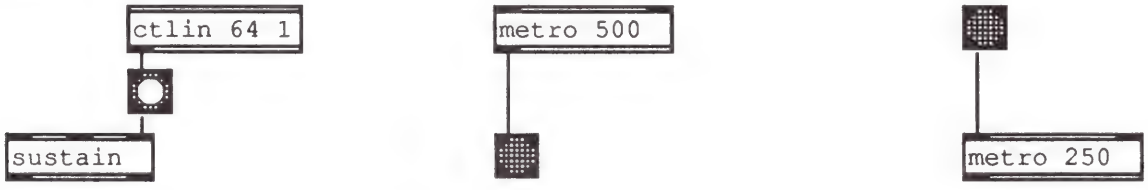
Arguments

The color of **led** can be specified by choosing the **Get Info...** command from the Max menu. A newly created **led** displays in red on a color monitor.

Output

int — The output is 1 when **led** is bright, 0 when it is dark. A bang in the inlet flashes **led** on and off and sends 0 out the outlet.

Examples



Displays an on/off state, announces activity with a flash, or can be used as a toggle

See Also

button	Flash on any message, send a bang
TogEdge	Report a change in zero/non-zero values
toggle	Switch between on and off (1 and 0)
Tutorial 40	Automatic actions

Input

dump — In left inlet: With no arguments, dump sends the entire bank of patches (if its been read or loaded into the **lib** object) out the object's outlet as a system exclusive bulk dump message. Followed by a number specifying a patch to send, dump sends a system exclusive bulk dump for one patch.

int — In left inlet: The numbers are individual bytes of a MIDI stream from **sysexin** or **midiin**. The **lib** object will respond to incoming MIDI system exclusive messages by storing bulk dumps from a MIDI device if the bytes match the patterns specified in the **lib** script.

int — In right inlet: Sets the **lib**'s MIDI channel. This number is displayed in the library window and can also be used by incoming or outgoing messages in the **lib** script.

load — In left inlet: With no arguments, load uses the "get bank" script defined for the **lib** object and sends a system exclusive message requesting a bank of patches. Followed by a number specifying which patch to get, load sends a system exclusive message requesting a single patch. If the device responds to these messages, **lib** will capture the input and store the bank or patch in its memory, replacing any existing patches.

send — In left inlet: send, followed by a number specifying an index into the current patch for the location of a parameter, the optional word param (or p) and a number specifying a parameter number, and a number specifying a data value for the parameter, sends a parameter change message according to the format of the param message in the **lib** script.

put — In left inlet: put has the format and result as the send message but will also store the data value in the current patch.

set — In left inlet: set, followed by two numbers, sets the "argument variable" indexed by the first number (1-9) to the value of the second number. These argument variables can be used in is expressions in the **lib** script, in conjunction with parameter or mode changes. For example, after the sending message set 7 20, the value of argument variable $i7 would be 20. Note that some argument variables have predefined uses. $i7-$i9 are available for your use. set was intended to do things such as change the "internal" or "cartridge" setting of a MIDI device.

read — In left inlet: read opens a standard **Open...** dialog box to read in a file of patches into the **lib** object. Only files whose file type matches that specified in the **lib** script will be listed in the dialog box.

write — In left inlet: write opens a standard **Save As...** dialog box to save the current contents of the library in a file. The file will have the filetype specified in the **lib** script and can be opened by sending the read message to **lib**.

open — In left inlet: open opens the **lib** object's window for displaying and editing patch names. This window can also be opened by double-clicking on the **lib** object box.

lib

Script-driven patch librarian

Arguments

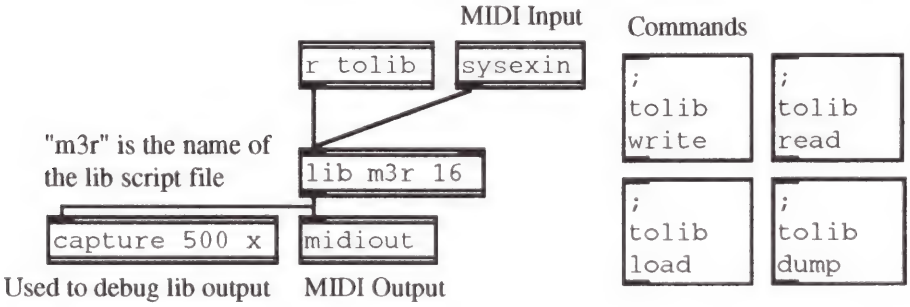
any symbol — Obligatory. The name of a script file which will be read in to define the behavior of the **lib** object. The script file can be located in the same directory as the Patcher file which contains the **lib** object, or in any folder in Max's search path.

int — Optional (follows the symbol). Sets the initial MIDI Channel. If no argument is present, the channel is 1 by default.

Output

int — **lib** outputs the bytes of MIDI system exclusive messages—patch dumps and parameter changes. The output of **lib** should be connected to **midiout**.

Examples



Typical configuation of **lib** *object*

See Also

Librarians	Creating patch editors and librarians
sysexin	Output incoming MIDI system exclusive messages
midiout	Transmit raw MIDI data
libfrom	Output stored patch value when editor is loaded
libto	Store data into an edited patch, send parameter changes
libswitch	Change the patch being edited within an editor window

Output stored patch value when editor is loaded

libfrom

Input

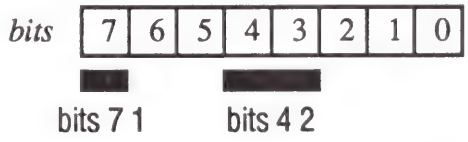
The **libfrom** object has no inlets. The data comes from a stored patch being edited in the Patcher window containing the **libfrom** object.

Arguments

(none) If no arguments are present, the **libfrom** object outputs the number of the patch being edited (the same number which is seen in the title bar of the Patcher window when it is opened from a **lib** object's library window).

int The **libfrom** object can have up to 16 arguments specifying offsets into a patch for sending stored values. The number of arguments determines the how many outlets the object will have. The outlets correspond to each argument from left to right.

bits If the word bits is present after an initial integer argument, you can specify that only certain bits at a particular location in the patch will be output. Following bits, two numbers are required: the first specifies the starting bit location, with 7 being the most significant bit and 0 being the least significant bit. The second number specifies how many bits to the right of the starting bit should be included. The diagram below shows an example.



When bits is used, only one patch offset can be specified, so the resulting **libfrom** object will have only one outlet.

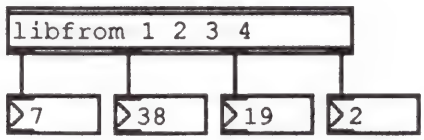
Output

int When a new patch is to be edited, each **libfrom** object in a Patcher window connected to **lib** outputs values of a patch stored in the library at patch locations specified by its arguments. The outlets of a **libfrom** object are usually connected to the inputs of user interface object such as a slider or Number box. If a **libfrom** object has multiple outlets, values are sent from right to left. The order in which one **libfrom** object outputs its values relative to another **libfrom** object is undefined. Use **buddy** to synchronize the outputs of two or more **libfrom** objects.

libfrom

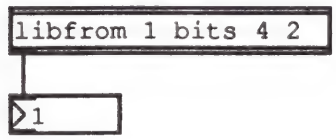
Output stored patch value when editor is loaded

Examples



Outputs values of patch locations 1-4

Outputs number of patch being edited in this window



Outputs bits 3 and 4 of patch location 2

See Also

Librarians	Creating patch editors and librarians
lib	Script-driven patch librarian
libto	Store data into an edited patch, send parameter changes
libswitch	Change the patch being edited within an editor window

Change the patch being edited within an editor window

libswitch

Input

int The number of a new patch in the library to be edited. The number should match the number that would be shown in the title bar of the editor window. In other words, you should take into account the first patch number (as defined in the patchnumbers statement in the **lib** script). When a valid patch number is received in its inlet, the **libswitch** object changes the title of the window of its patcher to the name of the new patch, then causes all of the **libfrom** objects in the editor patcher and its subpatchers to output the values contained in the new patch.

Arguments

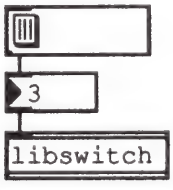
None

Output

The **libswitch** object has no outlets.

Examples

Note that the **libswitch** object will only work if is located in the Patcher named in the editor statement in the parent **lib** object's script. It will not work if it is contained in subpatchers of this Patcher.



Moving the slider will change the patch displayed and edited in the window. This is easier for the user than returning to the lib window and double-clicking on a different patch.

See Also

Librarians	Creating patch editors and librarians
lib	Script-driven patch librarian
libfrom	Output stored patch value when editor is loaded
libto	Store data into an edited patch, send parameter changes

libto

Store data into an edited patch, send parameter changes

Input

int — In all inlets: Numbers specify a value to be stored at an index in the patch being edited in the window containing the **libto** object. If the parent **lib** object's script file contains a param message, input to the **libto** object triggers a parameter change message sent out the outlet of the parent **lib** object.

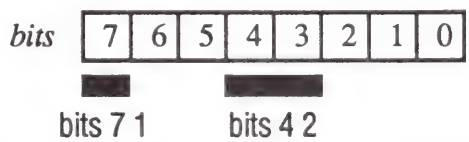
If the object contains more than one inlet, each inlet corresponds to the patch byte offset specified by the argument, from left to right.

When the Patcher window containing **libto** objects is first opened (or the patch being edited is changed) all of the **libto** objects are temporarily disabled so that the output of **libfrom** objects does not cause hundreds of unnecessary parameter changes to be sent.

Arguments

int — Up to 16 numbers specifying destination offsets within the current patch where incoming values will be stored. The number of arguments determine the number of inlets. When each inlet receives a number, it will store the number in the patch at the location specified by its corresponding argument.

bits — Follows the first (and only) number specifying the patch byte offset. bits can also follow a param argument descibed below. bits is followed by two numbers: the first specifies the starting bit location, with 7 being the most significant bit and 0 being the least significant bit. The second number specifies how many bits to the right of the starting bit should be included. The diagram below shows an example.



When bits is used, only one patch offset can be specified, so the resulting **libto** object will have only one inlet.

param (p) — Follows the first (and only) number specifying the patch index. param can also follow a bits argument described above. The word param, followed by a number, specifies a parameter number for the parameter change message to be sent when a number arrives in a **libto** inlet. If param is specified, the **libto** object can have only one inlet. param can be abbreviated with p.

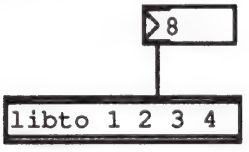
Output

The **libto** object has no outlets. However, if the **libto**'s parent **lib** object has a param message defined in its script, a parameter change message will be sent (as individual MIDI bytes) out the parent **lib** object's outlet.

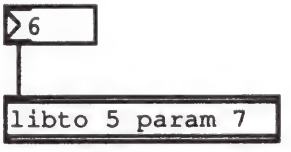
Store data into an edited patch, send parameter changes

libto

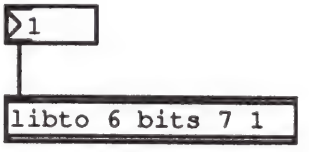
Examples



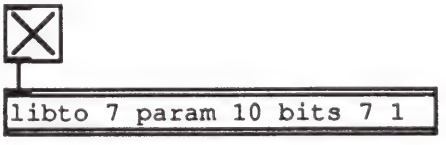
libto *object with multiple inlets. The Number box will store its output into index 3 in the current patch*



libto *object with single inlet that specifies the parameter number to be sent as part of a parameter change message*



libto *object with single inlet that sets only the most significant bit of index 6*



libto *object which specifies a parameter number and the bits to change at index 7 in the stored pacth*

See Also

Librarians	Creating patch editors and librarians
lib	Script-driven patch librarian
libfrom	Output stored patch value when editor is loaded
libswitch	Change the patch being edited within an editor window

line

Output numbers in a ramp from one value to another

Input

list The first number specifies a *target value*, and the second number specifies a total amount of time (in milliseconds). In that amount of time, numbers are output regularly in a line from the currently stored value to the target value.

int In left inlet: The number is the target value, to be arrived at in the time specified by the number in the middle inlet. If no time has been specified since the last target value, the time is considered 0 and **line** immediately outputs the target value.

In middle inlet: The number is the time, in milliseconds, in which to arrive at the target value.

In right inlet: The number is the interval (in milliseconds) at which intermediary numbers are regularly sent out.

Arguments

int Optional. The first argument sets an initial value to be stored in **line**. If there is no argument, the initial value is 0. The second argument sets an initial value for the *grain*, the time interval at which numbers are sent out. If the grain is not specified, **line** outputs a number every 20 milliseconds. The minimum grain allowed is 1 millisecond; any number less than 1 will be set to 20.

Output

int Out left outlet: Numbers are sent out at regular intervals, describing a straight line toward a target value. If a new target value and time are specified before the line is completed, the new line starts from the most recent output value, in order to avoid discontinuities.

If a value is received in the left inlet without an accompanying time value, it is sent out immediately (time is considered 0).

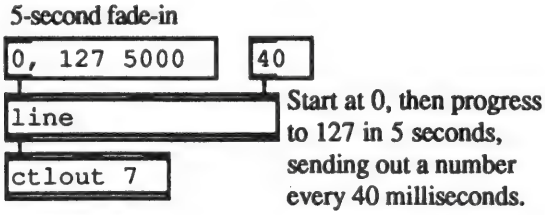
bang Out right outlet: When **line** has arrived at its target value, bang is sent out.

Note: In practice, the target value is arrived at in just under the amount of time specified (time minus grain).

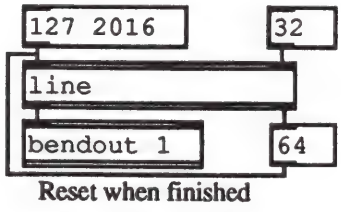
Output numbers in a ramp from one value to another

line

Examples


5-second fade-in
0, 127 5000
40
line
ctlout 7
Start at 0, then progress to 127 in 5 seconds, sending out a number every 40 milliseconds.

Output values in a straight line...



and bang *when finished*

See Also

Tutorial 31 Using timers

loadbang

Send a bang automatically when patch is loaded

Input

There are no inlets. Output is triggered automatically when the file is opened, or when the patch is part of another file that is opened.

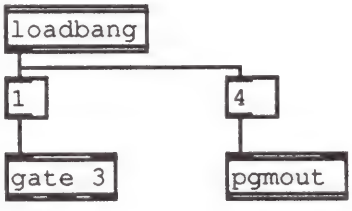
Arguments

None.

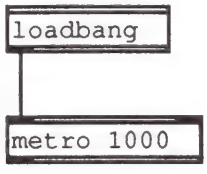
Output

bang Sent automatically when the patch is loaded.

Examples



Set inital values when a patch is loaded...



or start a process automatically

See Also

active	Send 1 when Patcher window is active, 0 when inactive
button	Flash on any message, send a bang
closebang	Send bang when Patcher window is closed
Tutorial 40	Automatic actions

Generate a note-off message following each note-on

makenote

Input

int — In left inlet: The number is treated as a pitch value for a MIDI note-on message. It is paired with a velocity value and the numbers are sent out the outlets. After a certain time, a note-off message (a note-on with a velocity of 0) is sent out for that pitch.

In middle inlet: The number is stored as a velocity to be paired with pitch numbers received in the left inlet.

In right inlet: The number is stored as the duration (in milliseconds) that **makenote** waits before a note-off message is sent out.

float — Converted to int.

list — The second number is treated as the velocity and is sent out the right outlet. The first number is treated as the pitch and is sent out the left outlet. A corresponding note-off message is sent out later.

stop — Causes **makenote** to send out immediate note-offs for all pitches it currently holds.

clear — Erases all notes currently held by **makenote**, *without* sending note-offs.

Arguments

int — Optional. The first argument sets an initial velocity value to be paired with incoming pitch numbers. If there is no argument, the initial velocity is 0. The second argument sets an inital note duration (time before a note-off is sent out), in milliseconds. If the second argument is not present, the note-off follows the note-on immediately.

float — Converted to int.

Output

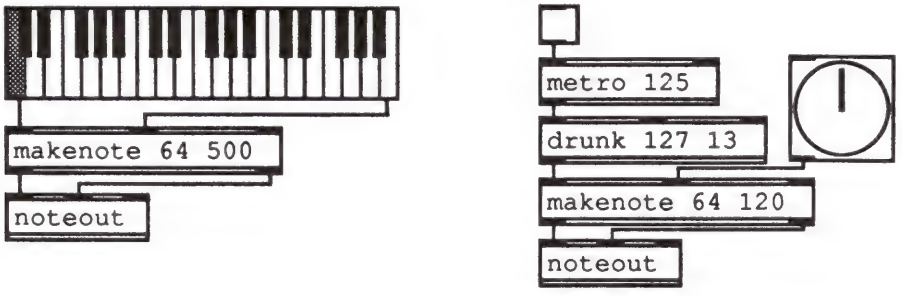
int — Out left outlet: The number received in the left inlet is sent out immediately, paired with a velocity value out the other outlet. After a certain duration, the same number is sent out paired with a velocity of 0.

Out right outlet: The number in the middle inlet is sent out as a velocity value in conjunction with a pitch value out the left outlet. After a certain duration, 0 is sent out paired with the same pitch.

makenote

Generate a note-off message following each note-on

Examples



Supply note-offs for note-ons generated within Max

See Also

stripnote	Filter out note-off messages, pass only note-on messages
xnoteout	Format MIDI note messages with release velocity
Tutorial 13	Managing note data

Look for a series of numbers, output it as a list

match

Input

int — If the numbers match the arguments, in the proper order, they are sent out as a list.

clear — Causes **match** to forget all numbers it has received up to that time.

set — The word set, followed by a list of numbers, specifies a new series of numbers **match** will look for.

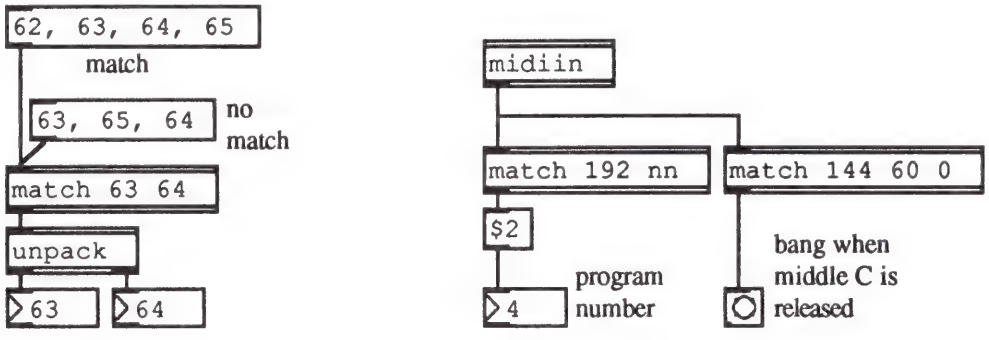
Arguments

list — Obligatory. The arguments specify numbers to look for, in the proper order. The word nn can be used as a *wild card* that will match any number.

Output

list — The numbers received in the inlet are compared with the arguments. If the numbers are the same, and in the same order, they are sent out the outlet as a list.

Examples



Numbers must be the same, and in the same order

See Also

iter — Break a list up into a series of numbers
pack — Combine numbers into a list
select — Select certain inputs, pass the rest on

maximum

Output the greatest in a list of numbers

Input

int — In left inlet: If the number is greater than the value currently stored in **maximum**, it is sent out the outlet. Otherwise, the stored value is sent out.

In right inlet: The number is stored for comparison with subsequent numbers received in the left inlet.

float — Converted to int, unless there is a float argument, in which case all numbers are compared as floats.

list — In left inlet: The numbers in the list are all compared to each other, and the greatest value is sent out the outlet. The value stored in **maximum** is replaced by the *next greatest* value in the list.

bang — In left inlet: Sends the most recent output out the outlet again.

Arguments

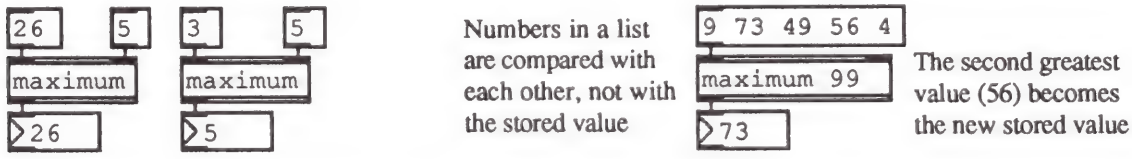
int or float — Optional. Sets an initial value to be compared with numbers received in the left inlet. If the argument contains a decimal point, all numbers are compared as floats, and the output is a float. If there is no argument, the initial value is 0.

Output

int — The number received in the left inlet is compared with the value currently held by **maximum** (or numbers received as a list are compared with each other), and the greatest of the numbers is sent out the outlet.

float — Only if there is an argument with a decimal point.

Examples



The output is the greater of two numbers, or the greatest in a list of numbers

See Also

past — Report when input increases beyond a certain number
Peak — If a number is greater than previous numbers, output it
> — *Is greater than*, comparison of two numbers

Pop-up menu, to display and send commands

menu

Input

int — The number specifies a **menu** item to be sent out, and causes **menu** to display that item. The items are numbered starting at 0.

A menu item can also be chosen from a **menu** as with any Macintosh pop-up menu.

bang — Sends out the currently displayed menu item.

set — The word set, followed by a number, specifies a menu item to be displayed by **menu**, but does not send it out the outlet.

setitem — The word setitem, followed by an item number and any message, sets the specifed menu item to that message.

append — The word append, followed by any message, appends that message as the last item in the menu.

Arguments

When a new **menu** object is created, a dialog box appears so that menu items can be typed in. A menu item can be any message.

If *Auto Size* is checked, the width of **menu**'s pop-up menu will be adjusted to fit the width of the longest item. If *Evaluate Item Text* is checked, the text of the selected menu item will be sent out the right outlet as a message.

To modify **menu**'s items or options, select the object (when the Patcher window is unlocked) and choose **Get Info...** from the Max menu.

Output

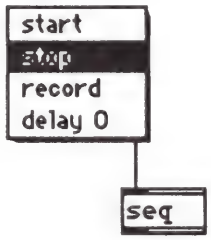
int — Out left outlet: The number of the selected menu item is sent out. Menu items are numbered beginning with 0.

anything — Out right outlet: If *Evaluate Item Text* has been checked, the text of the selected menu item is sent out as a message.

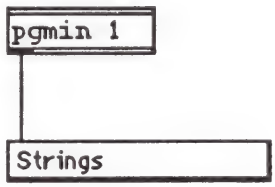
menu

Pop-up menu, to display and send commands

Examples



Used to send commands



...or to display text associated with numbers received

See Also

coll	Store a collection of different messages
Tutorial 37	Data structures

Send any message

message

Input

The **message** object is referred to in this manual as the **message box**.

bang — Sends out the contents of the message box. A mouse click on the message box has the same effect.

int or float — The number replaces the value stored in the argument $1, if such an argument exists, then sends out the contents of the message box.

list — Each item in the list is stored in place of its corresponding $ argument, if such an argument exisits, then the contents of the message box are sent out.

symbol — The word symbol, followed by a symbol, stores that symbol in the $1 argument, then sends out the contents of the message box.

set — The word set, followed by a message, sets the contents of the message box to that new message, without triggering output. The word set by itself erases the contents of the message box .

append — The word append, followed by a message, appends that message (preceded by a space) at the end of the contents of the message box, without triggering output.

Arguments

anything — The initial contents of the message box are typed in when the Patcher window is unlocked. Any message of up to 256 items can be contained in a message box. Certain characters have special meaning.

$ — A dollar sign ($), followed immediately by a number, is a changeable argument. This argument's value can be replaced by the corresponding item in a list received in the inlet. (Example: $2 stores the second item in a list as its value before sending out the contents of the message box.) The value of a changeable argument is initially 0.

, — A comma (,) divides a message into separate messages which will be sent out in order. (Example: 3, 4, 5 sends out 3, then 4, then 5.)

; — A semicolon (;) sends a message to a **receive** object. The first item following a semicolon is the name of the **receive** object. The rest of the message (or up to the next semicolon) is sent to that object, rather than out the outlet. The first item after the semicolon can be a changeable argument, so an incoming message can set the destination of the message "on the fly."

\ — A backslash (\) is used to negate the special traits of a special character. When a backslash immediately precedes a dollar sign, comma, or semicolon, the character is treated as a normal character. (Example: Notes played were C\, E\, and G.)

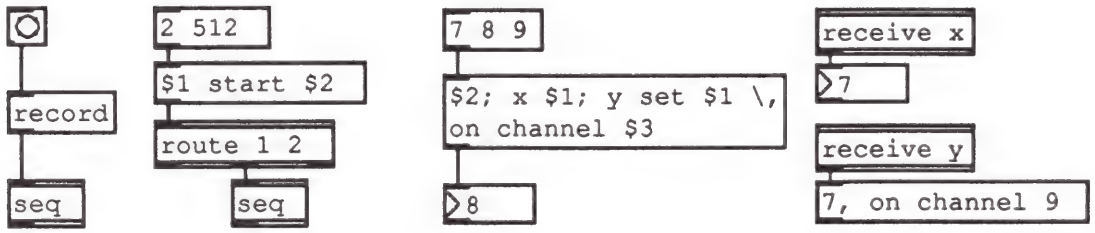
message

Send any message

Output

anything — The contents of the message box are normally sent out the outlet. If a semicolon is present, the rest of the message (or up to the next semicolon) is sent to the specified **receive** object, rather than out the outlet.

Examples



Send a simple message, or construct a message of any degree of complexity

See Also

append	Append arguments at the end of a message
prepend	Place one message at the beginning of another
receive	Receive messages without patch cords
Tutorial 1	Saying "Hello!"
Tutorial 25	Managing messages

Output bang,
at regular intervals

metro

Input

int — In left inlet: Any number other than 0 starts **metro**. At regular intervals, **metro** sends a bang out the outlet. 0 stops **metro**.

In right inlet: The number is the time interval, in milliseconds, at which **metro** sends out a bang. Any number less than 5 is set to 5. A new number in the right inlet does not take effect until the next output is sent.

float — Converted to int.

bang — In left inlet: Starts **metro**.

stop — In left inlet: Stops **metro**.

ext — In left inlet: The word ext, followed by a non-zero number, synchronizes **metro** to the MIDI Manager external clock. **metro** outputs bangs according to "intervals" of the external clock, so if no timing source is connected to Max's Clock In port, or Max is not using the MIDI Manager, **metro** will not output anything.

ext 0 sets **metro** to use Max's internal millisecond clock.

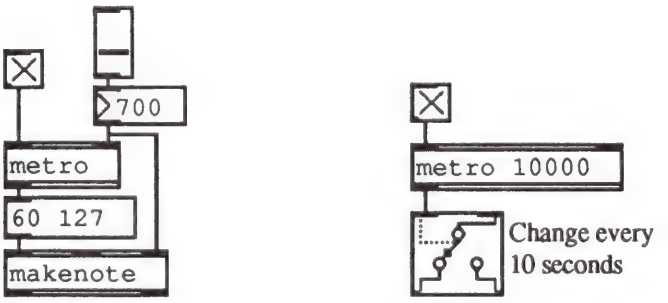
Arguments

int — Optional. The first argument sets an initial value for the time interval at which **metro** sends its output. If there is no argument, the initial time interval is 5 milliseconds. Any argument less than 5 will be set to 5. If the second argument is 1, **metro** uses the MIDI Manager external clock (see the ext message discussion above). If the second argument is 0 or not present, **metro** uses Max's internal millisecond clock.

Output

bang — A bang is sent immediately when **metro** is started, and at regular intervals thereafter.

Examples



Repeatedly send a message or trigger a process

metro

Output bang, at regular intervals

See Also

clocker	Report the elapsed time, at regular intervals
timein	Report time from external time code source
tempo	Output numbers at a metronomic tempo
Tutorial 4	Using **metro**

Send note-offs for hanging note-ons in raw MIDI data

midiflush

Input

int **midiflush** expects raw MIDI data from a source such as **seq** or **midiin**. **midiflush** echoes the data and observes which note-on messages on each channel have not received matching note-off messages.

bang When **midiflush** receives a bang, it outputs MIDI note-off messages for all note-ons which have not been matched by note-offs since the object was created (or the last bang message was sent).

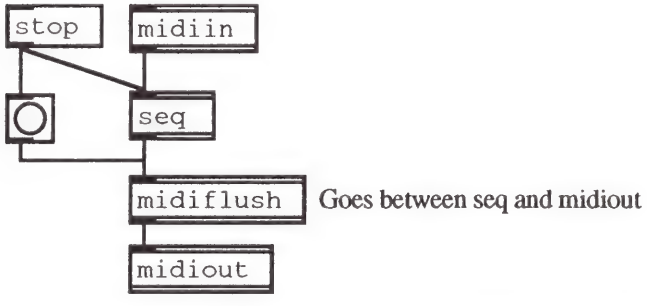
Arguments

None.

Output

int **midiflush** echoes all its input to its output, and sends MIDI note-off messages (as a series of numbers) for all note-ons which have not been matched by note-offs at its input.

Examples



When you click on stop, *the sequence stops and any hanging note-ons are matched when the bang message is sent to* **midiflush**

See Also

midiin Output incoming raw MIDI data
midiout Transmit raw MIDI data
seq Sequencer for recording and playing back MIDI

This page intentionally left blank

Prepare data in the form of a MIDI message

midiformat

Input

Numbers received in the inlets are used as data for MIDI messages. The data is formatted into a complete MIDI message (with the status byte determined by the inlet) and sent out the outlet as inidvidual bytes.

list In leftmost inlet: The first number is a pitch value and the second number is a velocity value, to be formatted into a note-on message.

In 2nd inlet: The first number is an aftertouch (pressure) value and the second number is a pitch value (key number), to be formatted into a polyphonic key pressure message.

In 3rd inlet: The first number is a control value and the second number is a controller number, to be formatted into a control message.

int In 4th inlet: The value is formatted into a program change message.

In 5th inlet: The value is formatted into an aftertouch (channel pressure) message.

In 6th inlet: The value is formatted into a pitch bend message.

In rightmost inlet: The number is stored as the channel number of the MIDI messages. The actual value of the status byte is dependent on the channel. Numbers greater than 16 are *wrapped around* to stay between 1 and 16.

float Converted to int.

Arguments

int Optional. Sets an initial value for the channel number of the MIDI messages. Numbers greater than 16 are *wrapped around* to stay between 1 and 16. If there is no argument, the channel number is initially set to 1.

float Converted to int.

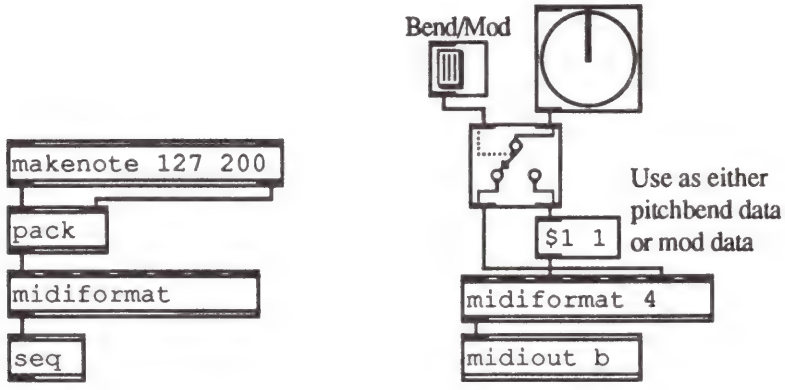
Output

int MIDI messages are sent out as individual bytes, for recording by the **seq** object or for transmission by the **midiout** object.

midiformat

Prepare data in the form of a MIDI message

Examples



Numbers are formatted into MIDI messages and sent out as individual bytes

See Also

MIDI	MIDI overview and specification
midiout	Transmit raw MIDI data
midiparse	Interpret raw MIDI data
Tutorial 34	Managing raw MIDI data

Output incoming raw MIDI data

midiin

Input

(MIDI) **midiin** receives all MIDI messages directly from the serial ports, or from the *virtual ports* of the MIDI Manager and/or OMS.

port The word port, followed by a letter a-z or the name of an OMS Input Device, sets the port from which the object receives incoming MIDI messages. The word port is optional and may be omitted.

Arguments

a-z Optional. Specifies the port from which to receive incoming MIDI messages. If there is no argument, **midiin** receives from port a.

If the MIDI Manager is being used, a letter a-f specifies a virtual port. If the MIDI Manager is not being used, the letter a specifies the modem port and b specifies the printer port. If OMS is being used, the letter a-z can be an abbreviation for any OMS or MIDI Manager port (configured using the **Midi Setup...** dialog).

Output

int All MIDI messages received from the specified port are sent out the outlet, byte-by-byte. Note that midin does not "clean up" any use of running status in the incoming MIDI stream.

Examples

MIDI messages received in a port are output by a **midiin** *object*

See Also

lib	Script-driven patch librarian
midiout	Transmit raw MIDI data
midiparse	Interpret raw MIDI data
rtin	Output incoming MIDI real time messages
sysexin	Output incoming MIDI system exclusive messages
xnotein	Interpret MIDI note messages with release velocity
xbendin	Interpret extra-precision MIDI pitch bend messages
MIDI	MIDI overview and specification
OMS	Using Max with OMS
Ports	How MIDI ports are specified
Tutorial 34	Managing raw MIDI data

midiout

Transmit raw MIDI data

Input

int — The number is transmitted as a byte of a MIDI message to the specified port.

float — Converted to int.

port — When not using OMS, the word port, followed by a letter a-f, specifies the port used to transmit the MIDI messages. When using OMS, port can be followed by a-z or the name of an OMS Output Device. In either case, the word port is optional and can be omitted.

Arguments

a-z — Optional. Specifies the port for transmitting MIDI data. If there is no argument, **midiout** transmits out port a.

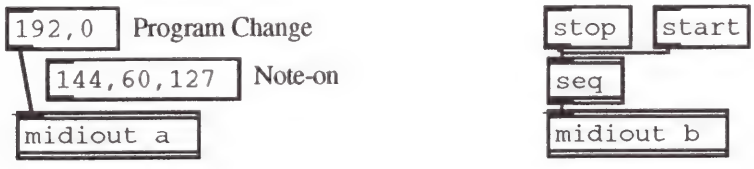
OMS name — Optional. When using OMS, the name of an OMS Output Device may be used as the first argument to specify the port.

If the MIDI Manager is being used, a letter a-f specifies a virtual port. If the MIDI Manager is not being used, the letter a specifies the modem port and b specifies the printer port. If OMS is being used, the letter a-z can be an abbreviation for any OMS or MIDI Manager port (configured using the **Midi Setup...** dialog).

Output

(MIDI) — There are no outlets. The output is a byte of a MIDI message transmitted directly to the serial ports, or to the *virtual ports*of the MIDI Manager or OMS.

Examples



MIDI bytes are received in the inlet and are transmitted out the specified port

See Also

lib	Script-driven patch librarian
midiin	Ouput incoming raw MIDI data
midiformat	Prepare data in the form of a MIDI message
MIDI	MIDI overview and specification
Tutorial 34	Managing raw MIDI data

Interpret raw MIDI data

midiparse

Input

int Numbers received in the inlet are treated as bytes of a MIDI message (usually from a **seq** or **midiin** object). The status byte determines the outlet which will be used to output the data bytes.

Output

list Out leftmost outlet: A note-on message. The first number is a pitch value and the second number is a velocity value.

Out 2nd outlet: A polyphonic key pressure message. The first number is an aftertouch (pressure) value and the second number is a pitch value (key number).

Out 3rd outlet: A control message. The first number is a control value and the second number is a controller number.

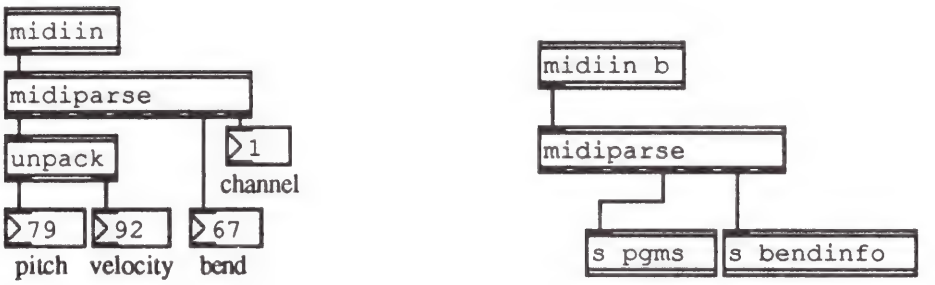
int Out 4th outlet: The number is a program change.

Out 5th outlet: The number is an aftertouch (channel pressure) value.

Out 6th outlet: The number is a pitch bend value.

Out rightmost outlet: The number is the MIDI channel number.

Examples



Interpret the meaning of MIDI messages and filter different types of data

See Also

midiformat	Prepare data in the form of a MIDI message
midiin	Output incoming raw MIDI data
Tutorial 34	Managing raw MIDI data

minimum

Output the smallest in a list of numbers

Input

int — In left inlet: If the number is less than the value currently stored in **minimum**, it is sent out the outlet. Otherwise, the stored value is sent out.

In right inlet: The number is stored for comparison with subsequent numbers received in the left inlet.

float — Converted to int, unless there is a float argument, in which case all numbers are compared as floats.

list — In left inlet: The numbers in the list are all compared to each other, and the smallest value is sent out the outlet. The value stored in **minimum** is replaced by the *next smallest* value in the list.

bang — In left inlet: Sends the most recent output out the outlet again.

Arguments

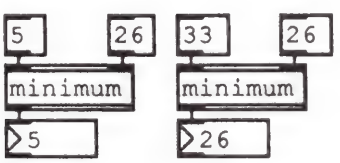
int or float — Optional. Sets an initial value to be compared with numbers received in the left inlet. If the argument contains a decimal point, all numbers are compared as floats, and the output is a float. If there is no argument, the initial value is 0.

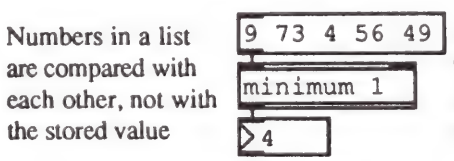
Output

int — The number received in the left inlet is compared with the value currently held by **minimum** (or numbers received as a list are compared with each other), and the smallest of the numbers is sent out the outlet.

float — Only if there is an argument with a decimal point.

Examples




Numbers in a list are compared with each other, not with the stored value
9 73 4 56 49
minimum 1
4
The second smallest value (9) becomes the new stored value

The output is the lesser of two numbers, or the smallest in a list of numbers

See Also

Trough — If a number is less than previous numbers, output it

< — *Is less than*, comparison of two numbers

Pass numbers only when the mouse button is up

mousefilter

Input

int If the mouse button is up, the number is sent out the outlet. Otherwise, the number is ignored.

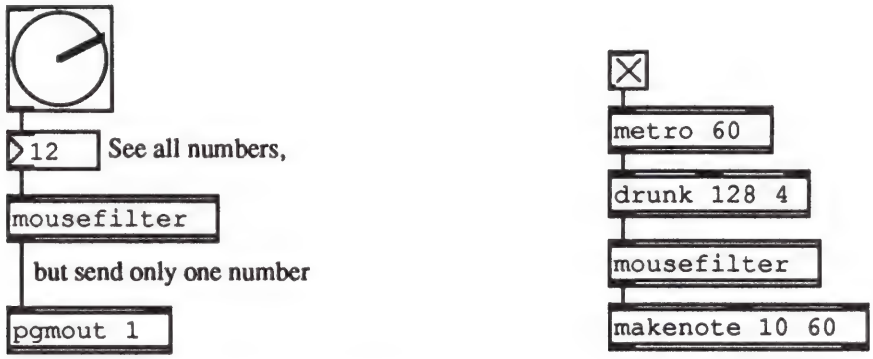
Arguments

None.

Output

int The number received in the inlet is sent out only if the mouse button is up.

Examples



Nothing gets through unless the mouse is up

See Also

MouseState Report the status and location of the mouse
Tutorial 39 Mouse control

MouseState

Report the status and location of the mouse

Input

bang — Sends out the current horizontal and vertical coordinates of the location of the mouse, as well as the change in location since the last output.

poll — Causes **MouseState** to send out the mouse location, and the change in mouse location, *whenever the mouse is moved*, as well as when a bang is received. If poll is followed by the name of a graphics window, the coordinates returned by **MouseState** will be local to the graphics window, and only sent while the graphics window is visible.

nopoll — Undoes a poll message, reverting **MouseState** to its normal condition of waiting for a bang before reporting.

zero — Resets the point **MouseState** considers as the 0,0 point from which to measure the mouse location. The current location of the mouse is considered the new 0,0 point.

reset — Resets the 0,0 point to its default setting, in the upper left corner of the screen.

Arguments

None.

Output

int — Out left outlet: Each time the mouse button is pressed, 1 is sent out. Each time the mouse button is released, 0 is sent out.

Out 2nd outlet: The horizontal location of the mouse, measured in terms of the number of pixels the mouse is to the right of the 0 point.

Out 3rd outlet: The vertical location of the mouse, measured in terms of the number of pixels the mouse is below the 0 point.

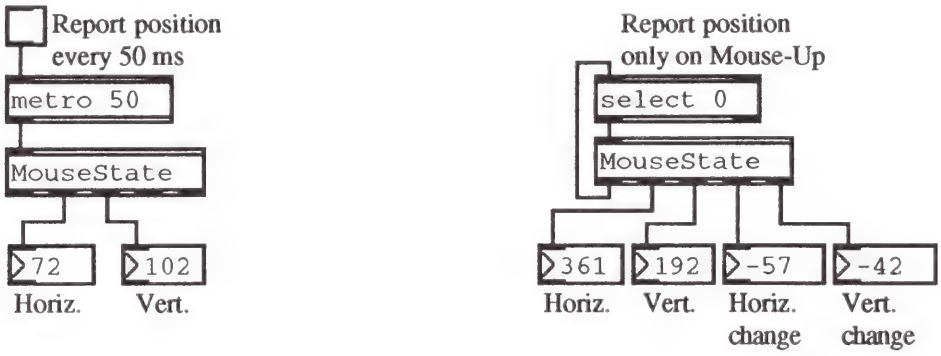
Out 4th outlet: The change in horizontal location of the mouse, since the last time the mouse location was reported.

Out right outlet: The change in vertical location of the mouse, since the last time the mouse location was reported.

Report the status and location of the mouse

MouseState

Examples



The mouse can provide continuous or discrete values

See Also

mousefilter Pass numbers only when the mouse button is up
Tutorial 39 Mouse control

msd

Load and send MIDI Sample Dump, read and write Sound Designer files

Input

int — In left inlet: Interpreted as a byte in a MIDI Sample Dump system exclusive message sequence received from **midiin** or **sysexin**.

In right inlet: Sets the MIDI channel (1-16) for transmitted MIDI Sample Dump messages.

load — In left inlet: The word load, followed by an optional number, sets the "sample number" to the number and sends a "dump request" message out the outlet. If the sampler begins sending data packets, **msd** accepts them until it receives the last packet, or until it receives the cancel message, or a MIDI Sample Dump cancel message.

send — In left inlet: The word send, followed by an optional number, sets the "sample number" and sends a "dump header" message out the outlet. If the sampler sends an acknowledge, **msd** begins sending data packets until it has sent all packets, or until it receives the cancel message, or a MIDI Sample Dump cancel message. If **msd** has not received or read in a sample, send does nothing.

read — In left inlet: Reads in a Sound Designer file (format I or II). A standard open file dialog is presented for the user to select a sampled sound file.

write — In left inlet: Writes a Sound Designer file (format I or II). A standard save file dialog is presented for the user to name the sampled sound file. If **msd** has not received or read in a sample, write does nothing.

cancel — In left inlet: Cancels any current Sample Dump reception or transmission activity, sending a MIDI Sample Dump cancel message out the outlet.

Arguments

int — Optional. Sets the initial MIDI Channel number for transmitting MIDI Sample Dump messages. The MIDI Channel can be changed via the right inlet. The default channel is 1.

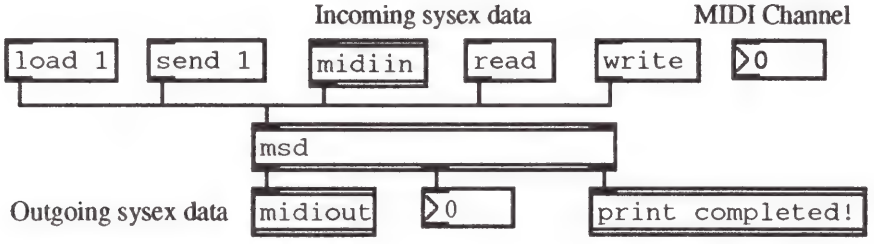
Output

int — Out left outlet: Individual bytes of MIDI Sample Dump system exclusive messages, intended to be sent to a **midiout** object.

Out middle outlet: The current packet number being sent or received.

bang — Out right outlet: Sent when a sample has been successfully sent or received.

Examples



Typical configuation for **msd** *with* **midiin** *and midiout* **objects**

See Also

midiin	Output incoming raw MIDI data
midiout	Transmit raw MIDI data

- MIDI Sample Dump specification (available from the International MIDI Association)
- Sound Designer File Format specification (available from Digidesign)

mtr

Multi-track sequencer

Input

record — In left inlet: Begins recording all messages received in the other inlets. The word record, followed by one or more track numbers, begins recording those tracks.

In other inlets: Begins recording messages on the track that corresponds to the inlet.

play — In left inlet: Plays back all messages recorded earlier, sending them out the corresponding outlets in the same rhythm and at the same speed they were recorded. The word play, followed by one or more track numbers, begins playing those tracks.

In other inlets: Plays back all messages on the track that corresponds to the inlet.

stop — In left inlet: Stops **mtr** when it is recording or playing. The word stop, followed by one or more track numbers, stops those tracks.

In other inlets: Stops the track that corresponds to the inlet.

next — In left inlet: Causes each track to output only the next message in its recorded sequence. When a next message is received, the track number and the delta time of each message being output are sent out the leftmost outlet as a list. The word next, followed by one or more track numbers, outputs the next message stored in that those tracks.

In other inlets: Outputs the next message stored on the track that corresponds to the inlet.

rewind — In left inlet: Resets **mtr** to the beginning of its recorded sequence. This command is used to return to the beginning of the sequence when stepping through messages with next. To return to the beginning of a sequence while playing or recording, just repeat the play or record message. When **mtr** is playing or recording, a stop message should precede a rewind message. The word rewind, followed by one or more track numbers, returns to the beginning of those tracks.

In other inlets: Returns the pointer to the beginning of the track that corresponds to the inlet.

mute — In left inlet: Causes **mtr** to stop producing output, while still continuing to "play" (still moving forward in the sequence). The word mute, followed by one or more tracks, mutes those tracks.

In other inlets: Mutes the track that corresponds to the inlet.

delay — In left inlet: The word delay, followed by a number of miliseconds, sets the first *delta time* value of each track to that number, so that all tracks begin playing back that amount of time after the play message is received.

In other inlets: Sets the initial delta time of the track that corresponds to the inlet.

first — In left inlet: The word first, followed by a number of milliseconds, causes **mtr** to wait that amount of time after a play message is received before playing back. Unlike delay, first does not alter the delta time value of the first event in a track, it just waits a certain time (in addition to the first delta time) before playing back from the beginning.

write — In left inlet: Calls up the standard **Save As...** dialog box, allowing the contents of **mtr** to be saved as a separate file. Note that the only way to save the contents of **mtr** is with the write message; the object's contents cannot be embedded in a Patcher file.

In other inlets: Writes a file containing only the track that corresponds to the inlet.

read — In left inlet: Calls up the standard **Open...** dialog box, so that a previously saved file can be read into **mtr**.

In other inlets: Opens a file containing only the track that corresponds to the inlet.

int — In any inlet other than the left inlet: If the track is currently being recorded, numbers received in that track's inlet are combined with a *delta time* (the number of milliseconds elapsed since the recording was started) and stored in **mtr**.

list — In any inlet other than the left inlet: If the track is currently being recorded, lists received in that track's inlet are stored in **mtr**, preceded by the delta time.

any symbol — In any inlet other than the left inlet: If the track is currently being recorded, symbols received in that track's inlet are stored in **mtr**, preceded by the delta time.

Although **mtr** can record individual bytes of MIDI messages received from **midiin**, it stores each byte with a separate delta time, and does not format the MIDI messages the way **seq** does. If you want to record complete MIDI messages and edit them later, **seq** is better suited for the task. On the other hand, **mtr** is perfectly suited for recording sequences of numbers, lists, or symbols from virtually *any* object in Max: specialized MIDI objects such as **notein** or **pgmin**, user interface objects such as **Number box**, **slider**, and **dial**, or any other object.

In order for a file to be read into **mtr** for playback, it must be in the proper format. An **mtr** multi-track sequence can even be typed in a text file, provided it adheres to the format. The contents of the different tracks are listed in order in an **mtr** file, and the format of each track is as follows. Note that a semicolon (;) ends each line.

Line 1:	track <track number>;	(Says which track to store subsequent data)
Line 2, etc.:	<delta time> <message>;	
Last line:	end;	(End of this track's data)

mtr

Multi-track sequencer

Arguments

int — Optional. Specifies the number of tracks in the **mtr**. The number of tracks determines the number of inlets and outlets *in addition to* the leftmost inlet and outlet. Up to 32 tracks are possible. If there is no argument, there will be only one track.

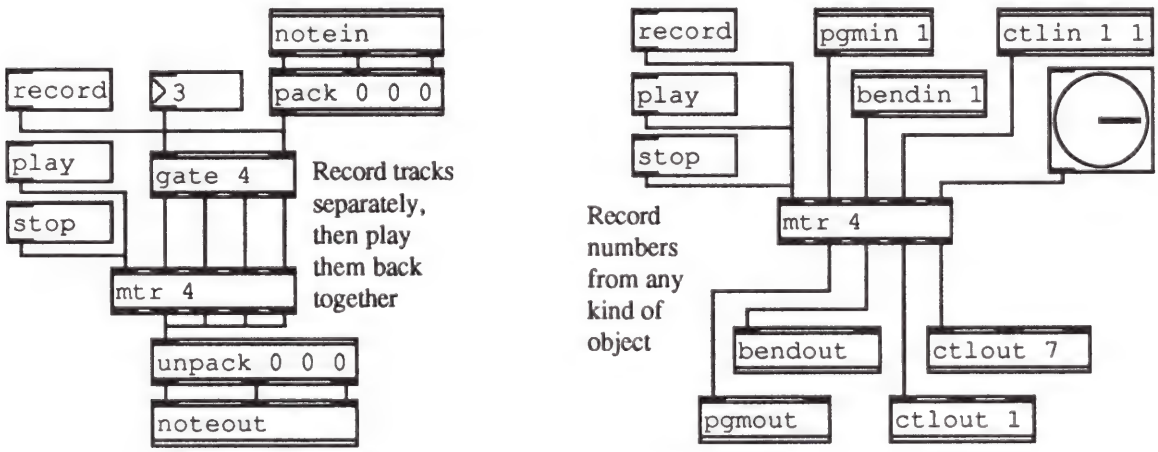
Output

anything — Out all track outlets: When a play message is received in the leftmost inlet, the messages stored in each track are sent out the outlet of that track, in the same rhythm and at the same speed they were recorded. A play message received in the inlet of an individual track plays that particular track.

When a next message is received in the leftmost inlet, the next message in each track is sent out its corresponding outlet. The word next, received in the inlet of an individual track, sends out the next message in that track.

list — Out left outlet: Whenever a value is sent out in response to a next message, the track number and delta time of that value are sent out the left outlet as a two-item list.

Examples



Record MIDI data or other events

See Also

seq	Sequencer for recording and playing MIDI
Sequencing	Recording and playing back MIDI performances
Tutorial 36	Multi-track sequencing

Multiple slider and scrolling display

multiSlider

Input

bang — Outputs the current slider values as a list.

border — The word border, followed by an integer, tells a **multiSlider** which of its outside borders to draw. This is useful for placing **multiSlider** objects next to each other. The arguments to border are:

border 0	Draw no borders
border 1	Draw left border
border 2	Draw right border
border 4	Draw top border
border 8	Draw bottom border

Any combination of borders can be drawn by adding these values. For example:

border 15	Draw all borders

(mouse clicks) — Clicking on any slider in a **multiSlider** immediately positions the slider at the click point. The current value of all sliders is output. Dragging across a **multiSlider** will set the other sliders in the same manner. If *continuous data* mode is enabled, the list of the current values will be sent out each time the mouse moves while dragging. If the *continuous data* mode is off, this list is only sent out when the mouse button is released.

Shift-clicking on a slider allows you to change a single slider without affecting any of the others, regardless of where you move the mouse.

Clicking on a *scrolling display* **multiSlider** has no effect.

cont_data — Toggles *continuous data* mode on and off. If this mode is turned on, the **multiSlider** will output a list of its current slider values each time the mouse moves during a dragging operation. If this mode is turned off, the **multiSlider** will only output a list when the mouse button is released.

float — Sets all slider values and positions to the number received and outputs a list reflecting the current values. If the **multiSlider** data type is set to int, the values in the incoming list are truncated and converted.

int — Sets all slider values and positions to the number received and outputs a list reflecting the current values. If the **multiSlider** data type is set to float, the values in the incoming list are converted to float.

list — Sets each slider to a corresponding value in the list from left to right, with the first value in the list setting the first slider. If the **multiSlider** has a different number of sliders than is present in the list, the number of sliders is changed to the number of items in the list. In such a case, the outside dimensions of the **multiSlider** will not change, only the width or height of the sliders.

max — Sets all sliders to their maximum values.

min — Sets all sliders to their minimum values.

peak_hold — Toggles *peak hold* mode on and off. When *peak hold* mode is on, the peak value of each slider is represented by a thin line.

peak_reset — Resets the current peak values to the current slider values.

select — Selectively sets slider values. For example, select 1 30 2 4 5 50 sets the first slider to 30, the second to 4, and the fifth slider to 50 (the top or leftmost slider is always number 1).

Arguments

You change a **multiSlider**'s modes and attributes by selecting it and choosing **Get Info...** from the Max menu. The attributes that can be set through the dialog box are:

- lower and upper limits (default -1.0 to 1.0)
- data type—int or float (default float)
- orientation of the sliders—horizontal or vertical (default vertical)
- number of sliders (default 1)
- display style—bars, thin line, or scrolling (default thin line)
- *peak hold* mode (default off)
- *continuous data* mode (default off)
- **multiSlider** borders (default all borders drawn)

Note: A scrolling display **multiSlider** may not be able to update at the rate it receives data. This will result in data points which are not displayed.

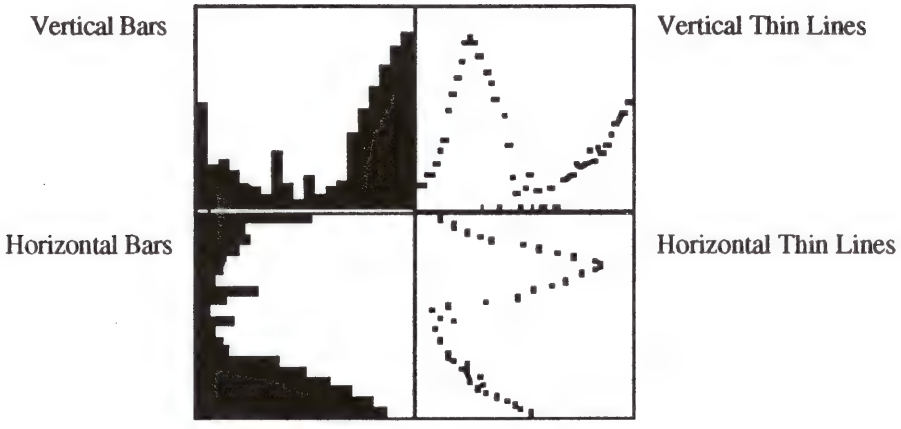
Output

list — When a **multiSlider** receives a list, int, or float in its inlet, it outputs a list of its current values. The list is also sent out when the sliders are changed with the mouse.

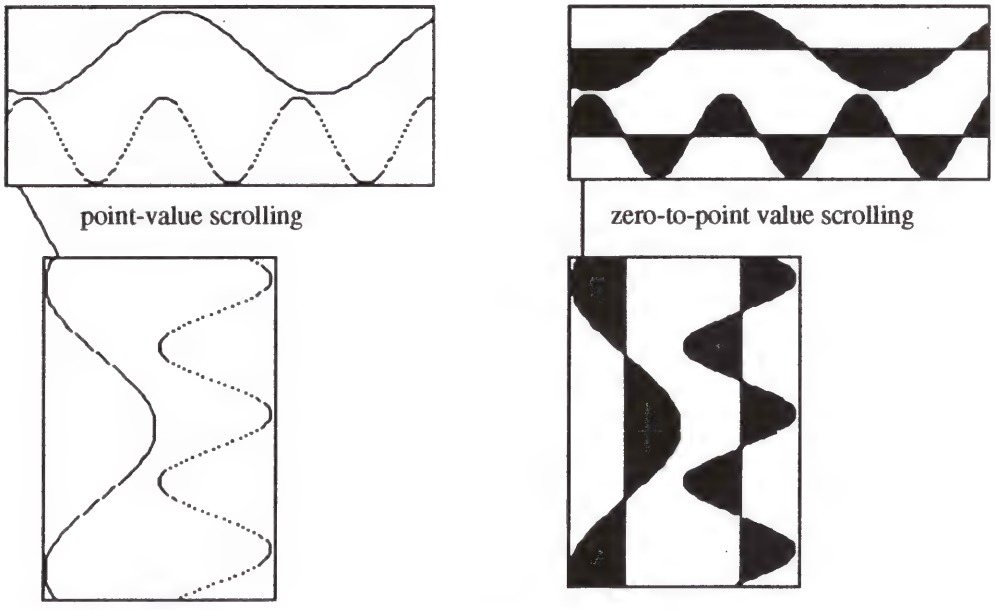
Multiple slider and scrolling display

multiSlider

Examples



multiSlider *drawing styles*



multiSlider *scrolling styles*

See Also

hslider	Output numbers by moving a slider onscreen
uslider	Output numbers by moving a slider onscreen
Tutorial 14	Sliders and dials

This page intentionally left blank

Output incoming MIDI note messages

notein

Input

(MIDI) **notein** receives its input from a MIDI note-on or note-off message received directly from the serial ports, or from the *virtual ports* of the MIDI Manager and/or OMS.

port The word port, followed by a letter a-z or the name of an OMS Input Device, sets the port from which the object receives incoming note messages. The word port is optional and may be omitted.

Arguments

a-z Optional. Specifies the port from which to receive incoming note messages. If there is no argument, **notein** receives from all channels on all ports.

OMS name Optional. When using OMS, you may use the name of an OMS Input Device as an argument to specify a port instead of a-z.

a-z and int A letter and number combination (separated by a space) indicates a port and a specific MIDI channel on which to receive note messages. Channel numbers greater than 16 will be *wrapped around* to stay within the 1-16 range.

int A number alone can be used in place of a letter and number combination. Numbers 1-16 specify a channel number on port a, numbers 17-32 specify channels 1-16 on port b, etc.

If the MIDI Manager is being used, a letter a-f specifies a virtual port. If the MIDI Manager is not being used, the letter a specifies the modem port and b specifies the printer port. If OMS is being used, the letter a-z can be an abbreviation for any OMS or MIDI Manager port (configured using the **MIDI Setup...** dialog).

Output

int Out left outlet: The number is the pitch value of the incoming note message.

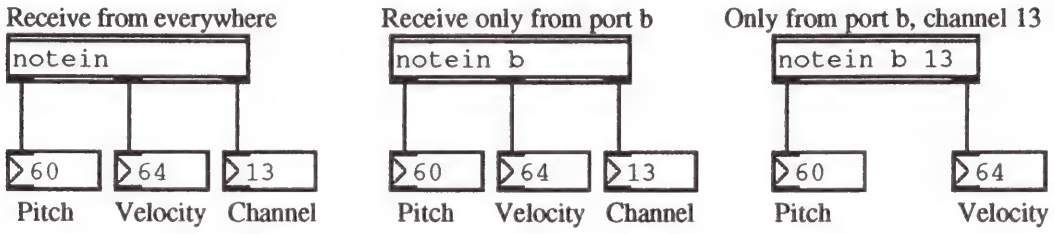
Out 2nd outlet: The number is the velocity of the incoming note-on message if non-zero, 0 for a note-off message. To receive release velocity, use **xnotein**.

If a specific channel number is included in the argument, there are only two outlets. If there is *no* channel number specified by the argument, **notein** will have a third outlet, on the right, which will output the channel number of the incoming note message.

notein

Output incoming MIDI note messages

Examples



Note-on messages can be received from everywhere, a specific port, or a specific port and channel

See Also

noteout	Transmit MIDI note messages
xnotein	Interpret MIDI note messages with release velocity
midiin	Output incoming raw MIDI data
OMS	Using Max with OMS
Ports	How ports are specified
Tutorial 12	Sending and receiving MIDI notes

Transmit MIDI
note messages

noteout

Input

int — In left inlet: The number is the pitch value of a MIDI note message transmitted on the specified channel and port. Numbers are limited between 0 and 127.

In middle inlet: The number is stored as the velocity of a note message, to be used with pitch values received in the left inlet. Numbers are limited between 0 and 127. 0 is considered a note-off message, 1-127 are note-on messages.

In right inlet: The number is stored as the channel number on which to transmit the note-on messages.

float — Converted to int.

list — In left inlet: The first number is used as the pitch, the second number is used as the velocity, and the third number is used as the channel, of a transmitted MIDI note message.

port — In left inlet: When not using OMS, the word port, followed by a letter a-f, specifies the port used to transmit MIDI note messages. When using OMS, port can be followed by a-z or the name of an OMS Output Device. In either case, the word port is optional and can be omitted.

Arguments

a-z — Optional. Specifies the port for transmitting MIDI note messages. Channel numbers greater than 16 received in the right inlet will be *wrapped around* to stay within the 1-16 range. If there is no argument, **noteout** initially transmits out port a, on MIDI channel 1.

a-z and int — A letter and number combination (separated by a space) indicates a port and a specific MIDI channel on which to transmit note messages. Channel numbers greater than 16 will be *wrapped around* to stay within the 1-16 range.

OMS name — Optional. When using OMS, the name of an OMS Output Device may be used as the first argument to specify the port.

int — A number alone can be used in place of a letter and number combination. Numbers 1-16 specify a channel number on port a, numbers 17-32 specify channels 1-16 on port b, etc. In this case, channel numbers received in the right inlet will specify both the port and the channel number.

If the MIDI Manager is being used, a letter a-f specifies a virtual port. If the MIDI Manager is not being used, the letter a specifies the modem port and b specifies the printer port. If OMS is being used, the letter a-z can be an abbreviation for any OMS or MIDI Manager port (configured using the **MIDI Setup...** dialog).

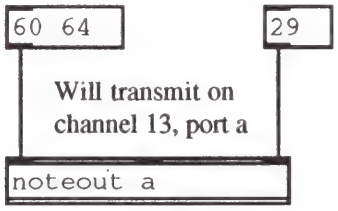
noteout

Transmit MIDI note messages

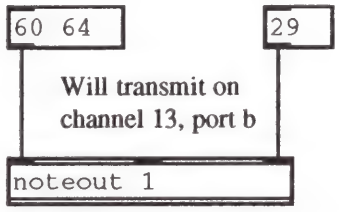
Output

(MIDI) There are no outlets. The output is a MIDI note-on message transmitted directly to the serial ports, or to the *virtual ports* of the MIDI Manager and/or OMS.

Examples



Letter argument transmits to only one port



Otherwise, number specifies both port and channel

See Also

notein	Ouput incoming MIDI note messages
xnoteout	Format MIDI note messages with release velocity
midiout	Transmit raw MIDI data
OMS	Using Max with OMS
Ports	How ports are specified
Tutorial 12	Sending and receiving MIDI notes

Display and output a number

Number box

Input

int or float — The number received in the inlet is stored and displayed in the **Number box** and sent out the outlet. A float is converted to int by an int **Number box**, and vice versa.

When the Patcher window is locked, numbers can be entered into the **Number box** by clicking on it with the mouse and typing in a number on the Macintosh keyboard. Typing the Return key or the Enter key, or clicking *outside* the **Number box**, sends the number out the outlet.

Dragging up and down on the **Number box** with the mouse (when the Patcher window is locked) moves the displayed value up and down, and outputs the new values continuously. In the float **Number box**, dragging to the left of the decimal point changes the value in increments of 1. Dragging to the right of the decimal point changes the fractional part of the number in increments of 0.01.

bang — Sends the currently displayed number out the outlet.

set — The word set, followed by a number, sets the stored and displayed value to that number without triggering output.

min — The word min, followed by a number, sets the minimum value that can be displayed by the **Number box**.

max — The word max, followed by a number, sets the maximum value that can be displayed by the **Number box**.

(typing) — When a **Number box** is *highlighted* (indicated by a filled-in triangle) in a Patcher window, numerical keyboard input is sent to the **Number box** to change its value. Clicking the mouse or pressing Return or Enter stores a pending typed number.

Arguments

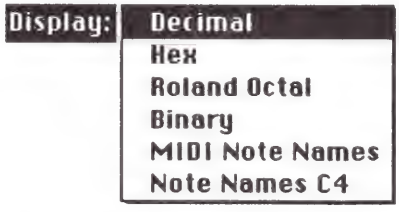
You can set minimum and maximum integer limits on stored, displayed, typed, and passed-through values by choosing **Get Info...** from the Max menu on a selected **Number box**. When the patch is loaded, the **Number box** is initially set to the minimum value, or to 0 if there is no minimum, or the minimum is less than 0. A newly created **Number box** has no minimum or maximum value.

Other options available in the **Get Info...** dialog box are: *Draw in Bold* (to display in bold typeface), *Draw Triangle* (to have an arrow pointing to the number, giving it a distinctive appearance), *Send on Mouse-Up* (to send a number only when the mouse button is released, rather than continuously), and *Can't Change* (to disallow changes with the mouse or the Macintosh keyboard).

In addition, you can choose among several ways of displaying numbers with the Display pop-up menu.

Number box

Display and output a number



Decimal is the normal method of displaying numbers. *Hex* shows numbers in hexadecimal, useful for MIDI-related applications. *Roland Octal* shows numbers in a format used by some user-primitive hardware devices where each digit ranges from 1 to 8. 11 is 0 and 88 is 63. *Binary* shows numbers as ones and zeroes. *MIDI Note Names* shows numbers according to their MIDI pitch value, with 60 displayed as C3. *Note Names C4* is the same as *MIDI Note Names* except 60 is displayed as C4.

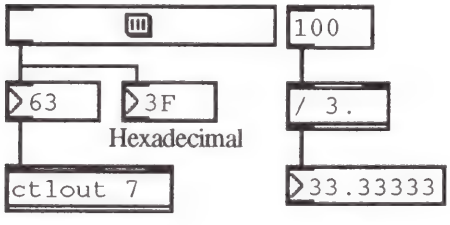
With all display modes, numbers must be typed in the format they are displayed.

The font and size of a **Number box** can be changed with the **Font** menu.

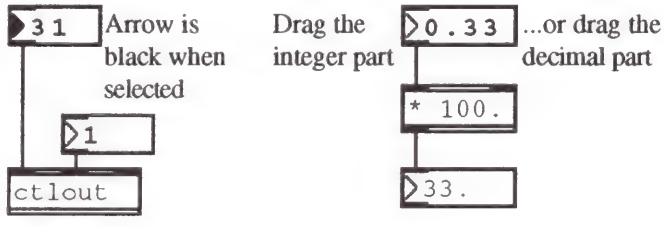
Output

int or float — The number displayed in the **Number box** is sent out the outlet. Numbers received in the inlet or typed on the Macintosh keyboard can exceed the limits of the **Number box**, but the value that is stored and displayed will automatically be limited to the specified range.

Examples



Displays numbers passing through



Can be used to output numbers

See Also

Tutorial 3 — About numbers
Tutorial 10 — Number boxes

Interpret numbers typed on the Macintosh keyboard

numkey

Input

int — The number is an ASCII value received from a **key** or **keyup** object. When digits are typed on the Macintosh keyboard, **numkey** recognizes the ASCII values and interprets them as the numbers being typed.

The keys recognized by **numkey** are the digits 0-9, the Delete (Backspace) key, decimal point (period), Return, and Enter. Digits are combined as a single number and stored in **numkey**.

bang — Sends the number currently stored in **numkey** out the left outlet, and resets the stored number to 0.

clear — Resets the stored number to 0.

Arguments

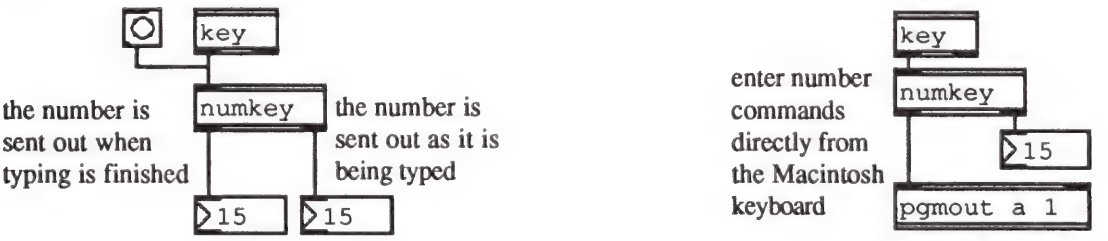
Optional. A float argument causes **numkey** to understand the decimal point and the fractional part of a number, and send out floats instead of ints.

Output

int — When digits are typed on the Macintosh keyboard, and the ASCII value (from **key** or **keyup**) is received in the inlet, the digits are combined as a single number and stored in **numkey**. The stored number is sent out the right outlet each time a new digit is typed. The Delete key erases the most recently typed digit, and sends the stored number out the right outlet. The period key acts as a decimal point and causes **numkey** not to store subsequent digits until a new number is started (unless there is a float argument). The Return or Enter key sends the stored number out the left outlet and resets the number stored in **numkey** to 0, so that a new number can be typed in.

float — When there is a float argument, **numkey** understands decimal points and fractional parts of a number, and sends out floats instead of ints.

Examples



Recognizes all numbers typed in

numkey

Interpret numbers typed on the Macintosh keyboard

See Also

key	Report keys typed on the Macintosh keyboard
keyup	Report keys released on the Macintosh keyboard
Number box	Display and output a number
Tutorial 20	Using the Macintosh keyboard

Set pop-up menu with names of OMS devices

omsinfo

Input

int — In left inlet: A number initiates a series of messages which set the individual items of a pop-up **menu** object connected to **omsinfo**'s outlet to the names of the current OMS *output* devices. The number sent to **omsinfo**'s left inlet is then sent in a set message to set the currently displayed menu item.

In right inlet: A number initiates a series of messages which set the individual items of a pop-up **menu** object connected to **omsinfo**'s outlet to the names of the current OMS *input* devices. The number sent to **omsinfo**'s right inlet is then sent in a set message to set the currently displayed menu item, unless the number is less than zero, in which case no set message is sent.

bang — In left inlet: Same as int, but doesn't send a set message after setting the menu items. The equivalent message to bang for input devices is -1 in the right inlet.

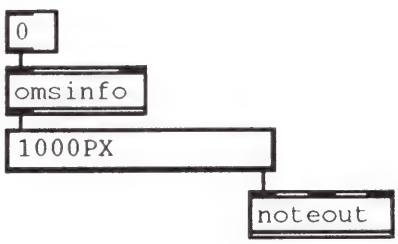
Arguments

None.

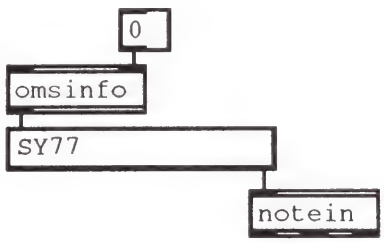
Output

(messages) — **omsinfo** first sends a clear message out its outlet which clears all the menu's items, followed by append messages for each OMS input or output device name. If the incoming message to **omsinfo** is an integer greater than or equal to zero, a set message is sent to set the currently displayed menu item.

Examples



omsinfo configuration for MIDI output objects



...and for MIDI input objects

See Also

menu	Pop-up menu, to display and send commands
Ports	How MIDI ports are specified
OMS	Using Max with OMS

This page intentionally left blank

Send messages out of a Patcher

outlet

Input

anything — Each **outlet** object in a patch will show up as an outlet at the bottom of an object box when the patch is used inside another patch (as an object or a subpatch). Messages received in the **outlet** object in the subpatch will come out of corresponding outlets in the subpatch's object box in the Patcher that contains it.

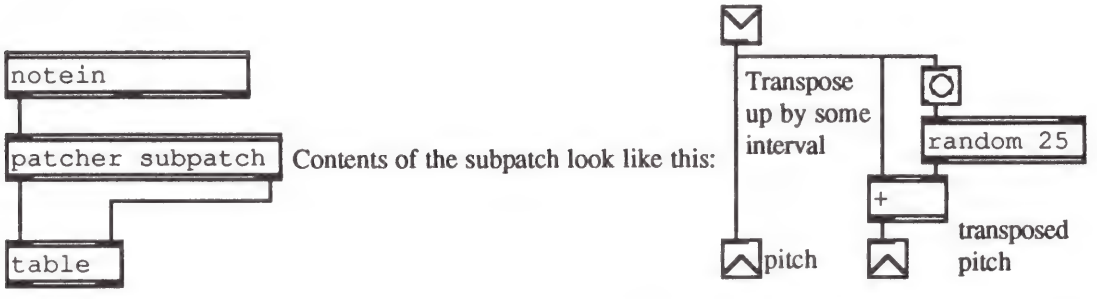
Arguments

A descriptive Assistance message can be assigned to an outlet by selecting the **outlet** object (when the Patcher window is unlocked) and choosing **Get Info...** from the Max menu. The Assistance message will appear when the patch is used inside another patch.

Output

anything — Any messages received by **outlet** in a subpatch are sent out the outlet of that subpatch, through patch cords.

Examples



Outlets of the subpatch object correspond to the **outlet** *objects inside the subpatch*

See Also

inlet	Receive messages from outside a Patcher
receive	Receive messages from other objects without patch cords
send	Send messages to other objects without patch cords
Tutorial 26	The **patcher** object

oval

Draw solid oval in graphics window

Input

bang — In left inlet: Draws the oval using the current screen coordinates, drawing mode, and color.

int — In left inlet: Sets the left screen coordinate of the oval and draws the shape.

In 2nd inlet: Sets the top screen coordinate of the oval.

In 3rd inlet: Sets the right screen coordinate of the oval.

In 4th inlet: Sets the bottom screen coordinate of the oval.

In 5th inlet: Sets the drawing mode of the oval. The following are drawing mode constants (refer to Inside Macintosh, pages I-157 and V- for the meaning of these constants).

Copy	0	blend	32
Or	1	addPin	33
Xor	2	addOver	34
Bic	3	subPin	35
NotCopy	4	transparent	36
NotOr	5	adMax	37
NotXor	6	subOver	38
NotBic	7	adMin	39

In 6th (right) inlet: Sets the palette index (color) of the oval according to the graphics window's current palette. This setting has no effect when the monitor is in black and white mode.

priority — In left inlet: The word priority, followed by a number greater than 0, sets an **oval** object's sprite priority in its graphics window. Objects with lower priority will draw behind those with a higher priority.

Arguments

any symbol — Obligatory. The first argument to **oval** must be the name of a graphics window into which the oval will be drawn. The window need not exist at the time the **oval** object is created, but the oval will not be drawn unless the name matches that of an existing and visible window.

int — Optional. Sets the initial sprite priority of the oval. If no priority is specified, the default is 3.

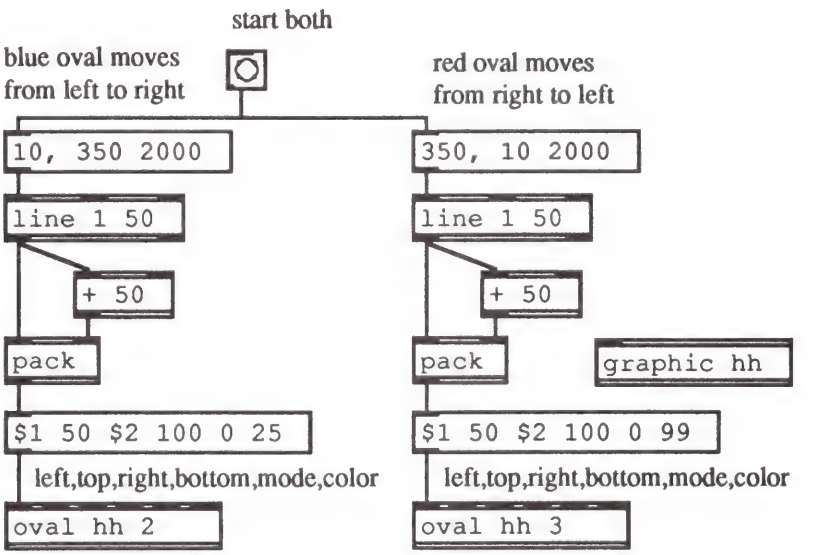
Draw solid oval in graphics window

oval

Output

(visual) When the **oval** object's associated graphics window is visible, and a bang message or a number is received in its left inlet, a shape is drawn in the window, and the object's previously drawn oval (if any) is erased.

Examples



*The **oval** object on the right will appear to pass in front of the one on the left when both move across the screen, since it has a higher sprite priority*

See Also

Graphics	Overview of Max graphics windows and objects
graphic	Graphics window object
frame	Draw framed rectangle in graphics window
rect	Draw solid rectangle in graphics window
ring	Draw framed oval in graphics window

pack

Combine numbers into a list

Input

int — The number received in each inlet is stored as an item in a list. A number in the left inlet causes the entire list to be sent out the outlet.

float — The number is converted to int, unless the inlet was initialized with a float argument.

bang — In left inlet: Outputs a list of the numbers currently stored.

list — In left inlet: Each number in the list is stored (limited by the number of inlets), and the stored list is sent out the outlet.

send — In left inlet: The word send, followed by the name of a **receive** object, sends a list of the currently stored numbers to all **receive** objects with that name, instead of out **pack**'s outlet.

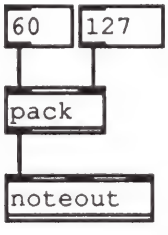
Arguments

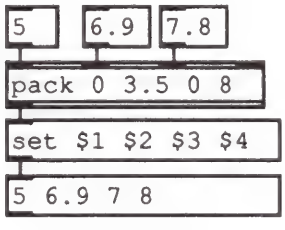
int or float — Optional. The number of inlets is determined by the number of arguments. Each argument sets an initial value for an item in the list stored by **pack**. If an argument contains a decimal point, that item will be stored as a float. The number of arguments is limited between 1 and 10. If there is no argument, there will be two inlets, and the two list items will be set to 0 initially.

Output

list — The length of the list is determined by the number of arguments. When input is received in the left inlet, the stored list is sent out the outlet.

Examples





See Also

swap — Reverse the sequential order of two numbers
thresh — Combine numbers into a list, when received close together
unpack — Break a list up into individual numbers
Tutorial 30 — Number groups

Report when input increases beyond a certain number

past

Input

list — The numbers in the list are compared to the arguments. If all of the numbers in the list are *greater than or equal to* the corresponding arguments, a bang is sent out the outlet. Before a bang is sent *again*, however, **past** must receive a clear message, or must receive another list in which the number that equalled or exceeded its argument goes back below (is less than) its argument.

int — If there is only one argument, and the input is *greater than or equal to* it, and the previous input was *not* greater than or equal to it, **past** sends a bang out the outlet.

float — Converted to int.

clear — Causes **past** to forget previously received input, readying it to send bang again.

set — The word set, followed by one or more numbers, sets the numbers which must be equalled or exceeded by the numbers received in **past**'s inlet.

Arguments

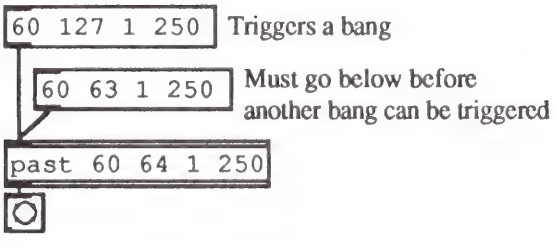
list — Sets the numbers which must be equalled or exceeded by the numbers received in the inlet.

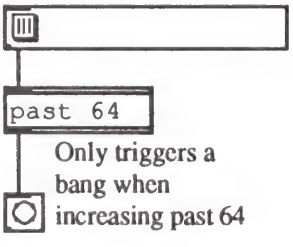
int — Sets a single number which must be equalled or exceeded by the number received in the inlet.

Output

bang — If the all of the arguments are equalled or exceeded by the numbers received in the inlet, **past** sends out a bang. Otherwise, **past** does nothing. A bang is sent only as a number *increases* past its threshold. Once the threshold has been passed, the number must go below the threshold again, then increase past it, before another bang will be sent.

Examples





Send out bang *only when the input goes past the threshold in an upward direction*

past

Report when input increases beyond a certain number

See Also

cd	Control CD Audio with Apple CD ROM driver
timeIn	Report time from external time code source
maximum	Output the greater of two numbers
Peak	If a number is greater than previous numbers, output it
>	*Is greater than,* comparison of two numbers

Create a subpatch within a patch

patcher / p

Input

anything — The number of inlets in a **patcher** object is determined by the number of **inlet** objects contained within its subpatch window.

Arguments

any symbol — Optional. The subpatch can be given a name by the argument, so that its name appears at the top of the subpatch window. The name at the top of the subpatch window is in brackets to indicate that it is part of another file. If there is no argument, the subpatch window is named [sub patch]. Different **patcher** objects which have the same name are distinct subpatches.

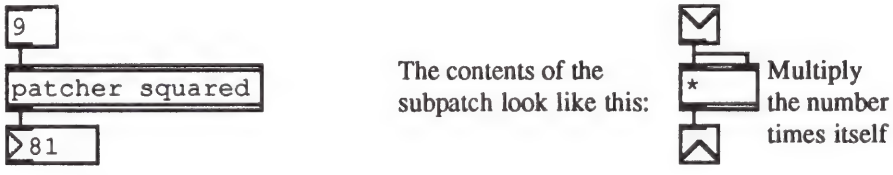
Output

anything — The number of outlets a **patcher** object has is determined by the number of **outlet** objects contained within the subpatch window. Output can also be sent via **send** and **value** objects contained in the subpatch. The actual messages sent out of a **patcher** object depend on the contents of the subpatch.

When a **patcher** object is first created, the subpatch window is automatically opened for editing. To view or edit the contents of a **patcher** object (or any subpatch object) later on, double-click on the object.

All the objects in a subpatch of a **patcher** object are saved in the Patcher which contains the object.

Examples



A patch can be contained (and saved) as part of another patch

See Also

Encapsulation	How much should a patch do?
inlet	Receive messages from outside a Patcher
outlet	Send messages out of a Patcher
Tutorial 26	The **patcher** object

pcontrol

Open and close subwindows within a Patcher

Input

open — Opens the Patcher window of any subpatches or **patcher** objects connected to **pcontrol**'s outlet.

close — Closes the Patcher window of any subpatches or **patcher** objects connected to **pcontrol**'s outlet.

enable — The word enable, followed by any number other than 0, enables the MIDI objects contained in the subpatches or **patcher** objects connected to **pcontrol**'s outlet. A message of enable 0 *disables* the MIDI objects in those subpatches.

help — The word help, followed by a symbol, opens a help file in Max's Help Directory with the name of the symbol followed by .help. This capability exists primarily for the use of the general Max help file (pat.help).

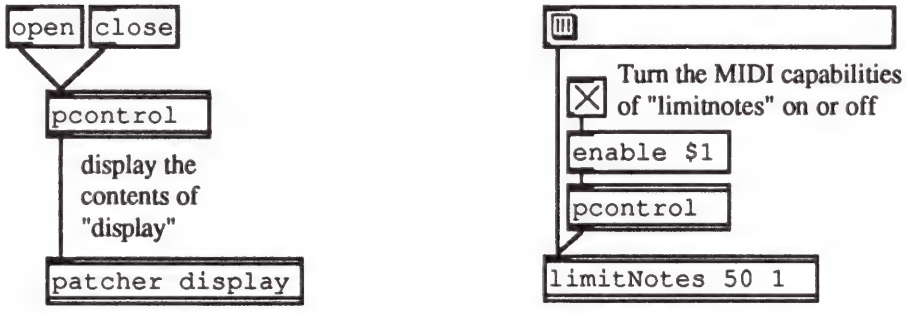
Arguments

None.

Output

Any subpatches or **patcher** objects connected to **pcontrol**'s outlet can have their Patcher window opened or closed, or MIDI enabled/disabled, when the appropriate message is received in the inlet of **pcontrol**.

Examples



Show/hide a subpatch window, or enable/disable its MIDI objects

See Also

inlet — Receive messages from outside a Patcher
Tutorial 40 — Automatic actions

If a number is greater than previous numbers, output it

Peak

Input

int — In left inlet: If the input is greater than the value currently stored in **Peak**, it is stored as the new peak value and is sent out.

In right inlet: The number is stored in **Peak** as the new peak value, and is sent out.

float — Converted to int.

list — In left inlet: The second number is stored as the new peak value and is sent out, then the first number is received in the left inlet.

Arguments

None. The initial value stored in **Peak** is 0.

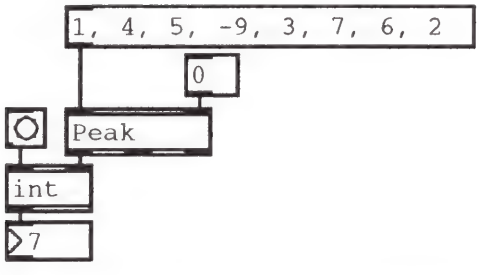
Output

int — Out left outlet: New peak values are sent out. (A number received in the right inlet is *always* the new peak value.)

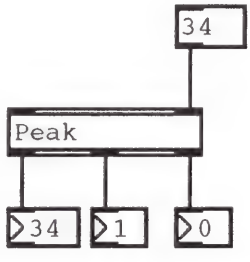
Out middle outlet: If the number received is a new peak value, the output is 1. If the number received in the left inlet is *not* a new peak value, the output is 0.

Out right outlet: If the number received is a new peak value, the output is 0. If the number received in the left inlet is *not* a new peak value, the output is 1.

Examples



Find the greatest in a series of numbers



A number in the right inlet always sets a new peak

See Also

maximum — Output the greater of two numbers
Trough — If a number is less than previous numbers, output it
> — *Is greater than,* comparison of two numbers

pgmin

Output incoming MIDI program change values

Input

(MIDI) **pgmin** receives its input from a MIDI program change message received directly from the serial ports, or from the *virtual ports* of the MIDI Manager and/or OMS.

port The word port, followed by a letter a-z or the name of an OMS Input Device, sets the port from which the object receives incoming program change messages. The word port is optional and may be omitted.

Arguments

a-z Optional. Specifies the port from which to receive incoming program change messages. If there is no argument, **pgmin** receives from all channels on all ports.

OMS name Optional. When using OMS, you may use the name of an OMS Input Device as an argument to specify a port instead of a-z.

a-z and int A letter and number combination (separated by a space) indicates a port and a specific MIDI channel on which to receive program change messages. Channel numbers greater than 16 will be *wrapped around* to stay within the 1-16 range.

int A number alone can be used in place of a letter and number combination. Numbers 1-16 specify a channel number on port a, numbers 17-32 specify channels 1-16 on port b, etc.

If the MIDI Manager is being used, a letter a-f specifies a virtual port. If the MIDI Manager is not being used, the letter a specifies the modem port and b specifies the printer port. If OMS is being used, the letter a-z can be an abbreviation for any OMS or MIDI Manager port (configured using the **MIDI Setup...** dialog).

Output

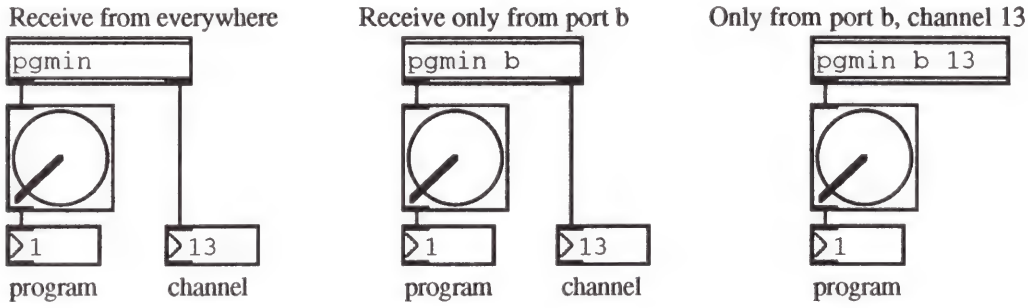
int If a specific channel number is included in the argument, there is only one outlet. The output is the incoming program number on the specified channel and port.

If there is no channel number specified by the argument, **pgmin** will have a second outlet, on the right, which will output the channel number of the incoming program change message.

Output incoming MIDI program change values

pgmin

Examples



Program changes can be received from everywhere, a specific port, or a specific port and channel

See Also

pgmout	Transmit MIDI program change messages
midiin	Output incoming raw MIDI data
Ports	How ports are specified
Tutorial 16	More MIDI ins and outs

pgmout

Transmit MIDI program change messages

Input

int — In left inlet: The number is transmitted as a program change value on the specified channel and port. Numbers are limited between 0 and 127.

In right inlet: The number is stored as the channel number on which to transmit the program change messages.

float — Converted to int.

list — In left inlet: The first number is the program number, and the second number is the channel, of a MIDI program change message, transmitted on the specified channel and port.

port — In left inlet: When not using OMS, the word port, followed by a letter a-f, specifies the port used to transmit the program change messages. When using OMS, port can be followed by a-z or the name of an OMS Output Device. In either case, the word port is optional and can be omitted.

Arguments

a-f — Optional. Specifies the port for transmitting MIDI program change messages. Channel numbers greater than 16 received in the right inlet will be *wrapped around* to stay within the 1-16 range. If there is no argument, **pgmout** initially transmits out port a, on MIDI channel 1.

a-f and int — A letter and number combination (separated by a space) indicates a port and a specific MIDI channel on which to transmit program change messages. Channel numbers greater than 16 will be *wrapped around* to stay within the 1-16 range.

OMS name — Optional. When using OMS, the name of an OMS Output Device may be used as the first argument to specify the port.

int — A number alone can be used in place of a letter and number combination. Numbers 1-16 specify a channel number on port a, numbers 17-32 specify channels 1-16 on port b, etc. In this case, channel numbers received in the right inlet will specify both the port and the channel number.

If the MIDI Manager is being used, a letter a-f specifies a virtual port. If the MIDI Manager is not being used, the letter a specifies the modem port and b specifies the printer port. If OMS is being used, the letter a-z can be an abbreviation for any OMS or MIDI Manager port (configured using the **MIDI Setup...** dialog).

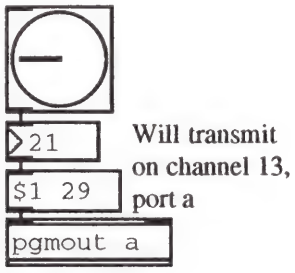
Output

(MIDI) — There are no outlets. The output is a MIDI program change message transmitted directly to the serial ports, or to the *virtual ports* of the MIDI Manager and/or OMS.

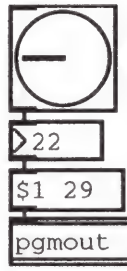
Transmit MIDI program change messages

pgmout

Examples



Will transmit on channel 13, port a



Will transmit on channel 13, port b

Letter argument transmits to only one port.
Otherwise, number specifies both port and channel

See Also

pgmin	Ouput incoming MIDI program change values
midiout	Transmit raw MIDI data
Ports	How ports are specified
Tutorial 16	More MIDI ins and outs

pics

Animation in graphics window

Input

int — Draws the specified frame number if the object's associated graphics window is visible. In a **pics** object, the first frame number is always 0. In addition, the "next frame" pointer is set to the frame number drawn.

bang — Draws the next frame. **pics** will cycle back to the first frame after drawing its last frame.

priority — The word priority, followed by a number greater than 0, sets the object's *sprite priority* to that number. Refer to the Graphics section of *Topics* for a discussion of sprite priorities.

Arguments

pics2 — Optional. If the object is created as **pics2**, the first frame is taken to be a background frame which is not erased by subsequent frames. Since the PICS file format is ambiguous on what the first frame does, you should try creating the object as **pics2** if you don't get the proper results with **pics**.

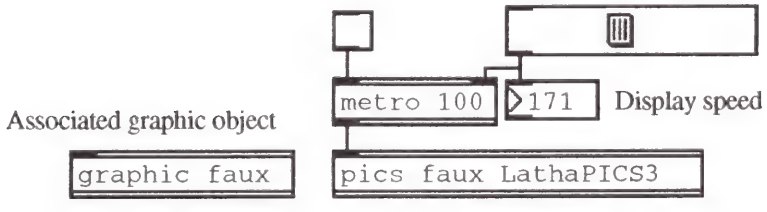
symbol — Obligatory. The first argument to **pics** must be a name of a **graphic** object whose window will be used to draw the animation. The second argument must be the name of a file in PICS format which will be loaded when the object is created. Note that the size of PICS files is often quite large, and loading the file may take a while.

int — Optional. Following the window name and file name, a number greater than 0 sets the inital sprite priority. The default priority is 3.

Output

(visual) — When the **pics** object's associated graphics window is visible, and a bang message or a number is received in its inlet, a frame is drawn in the window, and the object's previously drawn frame (if any) is erased, unless the object was created as **pics2**, in which case the first frame always stays on the screen.

Examples



Using a metronome to cycle through a PICS file at a constant speed.

See Also

graphic	Graphics window object
Graphics	Overview of graphics windows and objects
oval	Draw shape in graphics window
pict	Draw picture in graphics window

pict

Draw picture in graphics window

Input

bang — Draws the picture stored in the **pict** object if its associated graphics window is visible.

priority — The word priority, followed by a number greater than or equal to 0, sets the object's *sprite priority* to that number. Refer to the Graphics section in *Topics* for a discussion of sprite priorities.

Arguments

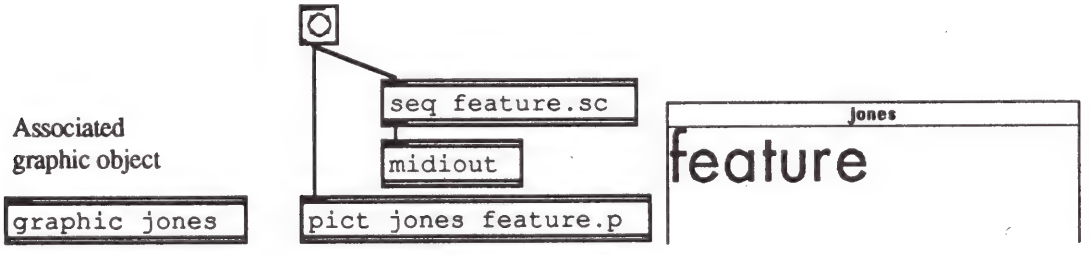
symbol — Obligatory. The first argument to **pict** must be a name of a **graphic** object whose window will be used to draw the picture. The second argument must be the name of a PICT file which will be loaded when the object is created.

int — Optional. Following the window name and file name, a number greater than or equal to 0 sets the inital sprite priority. The default priority is 0, which means the picture would be drawn behind all other objects.

Output

(visual) — When the **pict** object's associated graphics window is visible, and a bang message is received in its inlet, a picture is drawn in the window.

Examples



One bang starts the sequence and displays the picture.

See Also

graphic — Graphics window object
Graphics — Overview of graphics windows and objects
oval — Draw solid oval in graphics window
pics — Animation in graphics window

Delay numbers or lists

pipe

Input

int — In left inlet: The number is delayed a certain number of milliseconds before it is sent out the left outlet. If there are middle inlets, the numbers in those inlets are also delayed and sent out the corresponding outlets.

In right inlet: The number is stored as the time (in milliseconds) to delay numbers received in the other inlets.

bang — In left inlet: Retriggers the numbers currently stored in the **pipe** to be output again in the specified number of milliseconds (*in addition to* the numbers already being delayed).

float — Converted to int.

list — In left inlet: Numbers are distributed to **pipe**'s inlets to be delayed together. If there is a number for the right inlet, it sets the delay time for the other numbers.

clear — In left inlet: Halts all numbers currently being delayed by **pipe**.

Arguments

int — Optional. The last argument sets an initial value for the delay time, in milliseconds. If there is no argument, the delay time is 0. If there are two arguments, the first argument sets an initial value to be stored in **pipe**, and the second arguments sets the delay time. If more than two arguments are present, **pipe** creates additional inlets and outlets for delaying additional numbers in parallel to the leftmost one.

float — The last argument is converted to int. Other float arguments cause the corresponding outlet to send a float.

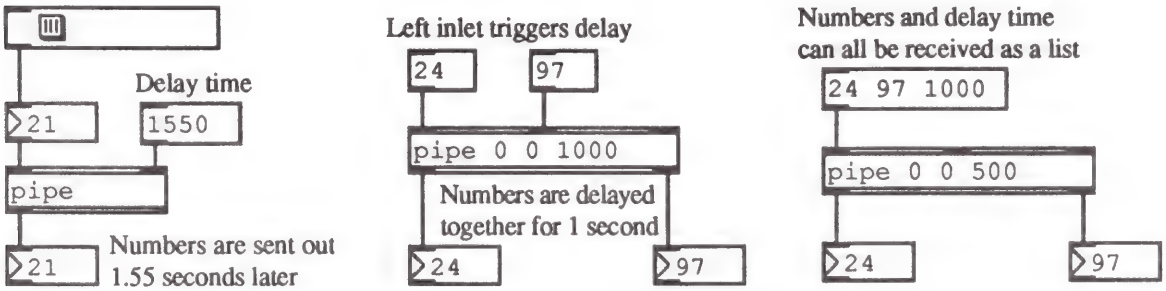
Output

int — When a number is received in **pipe**'s left inlet, it is delayed by the time specified, then sent out the left outlet. If there are middle inlets, the numbers in those inlets are also delayed and sent out their corresponding outlet, in response to a number is received in the left inlet. Unlike **delay**, more than one number at a time can be delayed in a **pipe**. When a new delay time is received in the right inlet, it does not affect the numbers already being delayed by **pipe** will come out.

pipe

Delay numbers or lists

Examples



One or more numbers can be delayed with **pipe**

See Also

delay	Delay a bang before passing it on
Tutorial 22	Delay lines

Allocate notes to different voices

poly

Input

list — In left inlet: The first number is treated as a pitch, and the second number is treated as a velocity value, of a pitch-velocity pair. If the velocity is not 0, **poly** allocates that note-on to the first available voice number and sends it out. If the velocity is 0, **poly** frees the voice that is holding that pitch and sends out the note-off.

int — In left inlet: The number is treated as the pitch value of pitch-velocity pair and the note is sent out.

In right inlet: The number is stored as the velocity to be paired with numbers received in the left inlet.

float — Converted to int.

stop — In left inlet: Immediately sends note-offs for all the notes currently being held by **poly**, freeing all voices.

Arguments

int — Optional. The first argument sets the number of voices to which **poly** can allocate notes (thus limiting the number of notes **poly** can hold at one time). If there is no argument present, **poly** can hold 16 notes.

If there is no second argument, or if the second argument is 0, **poly** sends any notes it cannot hold out the rightmost outlet. If there *is* a second argument not equal to 0, **poly** *steals voices*: when **poly** receives more notes than it has voices, it turns off the note it has held the longest and puts the new note in its place.

float — Converted to int.

Output

int — Out left outlet: The output is the voice number of the note-on or note-off being sent out.

Out 2nd outlet: The output is the pitch of the note-on or note-off.

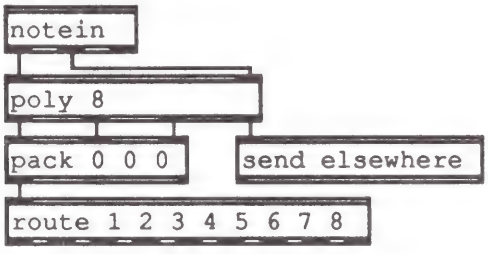
Out 3rd outlet: The number is the velocity of the note-on or note-off.

list — Out 4th outlet: The first number is the pitch, and the second number is the velocity, of any notes **poly** cannot hold. If there is a second argument, **poly** steals voices rather than send out overflow, and the fourth outlet is not created.

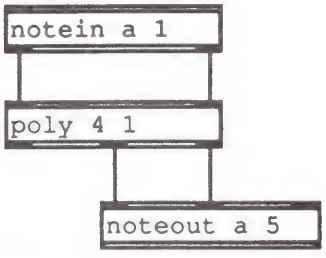
poly

Allocate notes to different voices

Examples



Send each voice to a different place



Limit the number of notes held at a time

See Also

flush Provide note-offs for held notes
makenote Generate a note-off message following each note-on

Output incoming MIDI poly pressure values

polyin

Input

(MIDI) **polyin** receives its input from MIDI polyphonic key pressure message received directly from the serial ports, or from the *virtual ports* of the MIDI Manager.

port The word port, followed by a letter a-z or the name of an OMS Input Device, sets the port from which the object receives incoming polyphonic key pressure messages. The word port is optional and may be omitted.

Arguments

a-f Optional. Specifies the port from which to receive incoming MIDI messages. If there is no argument, **polyin** receives from all channels on all ports.

OMS name Optional. When using OMS, you may use the name of an OMS Input Device as an argument to specify a port instead of a-z.

a-z and int A letter and number combination (separated by a space) indicates a port and a specific MIDI channel on which to receive polyphonic key pressure messages. Channel numbers greater than 16 will be *wrapped around* to stay within the 1-16 range.

int A number alone can be used in place of a letter and number combination. Numbers 1-16 specify a channel number on port a, numbers 17-32 specify channels 1-16 on port b, etc.

If the MIDI Manager is being used, a letter a-f specifies a virtual port. If the MIDI Manager is not being used, the letter a specifies the modem port and b specifies the printer port. If OMS is being used, the letter a-z can be an abbreviation for any OMS or MIDI Manager port (configured using the **MIDI Setup...** dialog).

Output

int Out left outlet: The number is the pressure value of the incoming polyphonic key pressure message.

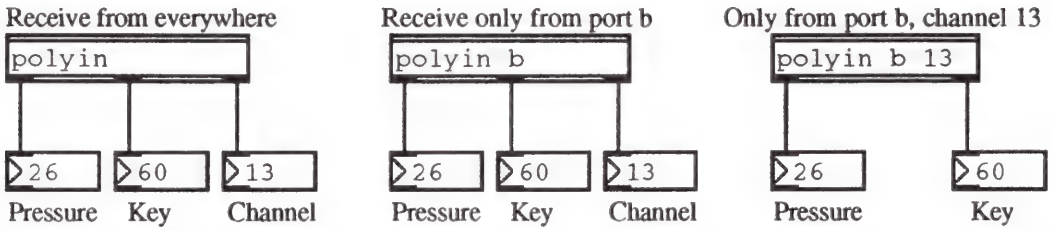
Out 2nd outlet: The number is the pitch value (key number) of the incoming message.

If a specific channel number is included in the argument, there are only two outlets. If there is *no* channel number specified by the argument, **polyin** will have a third outlet, on the right, which will output the channel number of the incoming note-on message.

polyin

Output incoming MIDI poly pressure values

Examples



Messages can be received from everywhere, a specific port, or a specific port and channel

See Also

polyout	Transmit MIDI poly pressure messages
midiin	Output incoming raw MIDI data
OMS	Using Max with OMS
Ports	How ports are specified
Tutorial 16	More MIDI ins and outs

Transmit MIDI poly pressure messages

polyout

Input

int — In left inlet: The number is the pressure value of a MIDI polyphonic key pressure message transmitted on the specified channel and port. Numbers are limited between 0 and 127.

In middle inlet: The number is stored as the key number, to be used with pressure values received in the left inlet. Numbers are limited between 0 and 127.

In right inlet: The number is stored as the channel number on which to transmit the polyphonic key pressure messages.

float — Converted to int.

list — In left inlet: The first number is the pressure value, the second number is the key number, and the third number is the channel, of a transmitted MIDI polyphonic key pressure message.

port — In left inlet: When not using OMS, the word port, followed by a letter a-f, specifies the port used to transmit the MIDI messages. When using OMS, port can be followed by a-z or the name of an OMS Output Device. In either case, the word port is optional and can be omitted.

Arguments

a-z — Optional. Specifies the port for transmitting MIDI polyphonic key pressure messages. Channel numbers greater than 16 received in the right inlet will be *wrapped around* to stay within the 1-16 range. If there is no argument, **polyout** initially transmits out port a, on MIDI channel 1.

a-z and int — A letter and number combination (separated by a space) indicates a port and a specific MIDI channel on which to transmit polyphonic key pressure messages. Channel numbers greater than 16 will be *wrapped around* to stay within the 1-16 range.

OMS name — Optional. When using OMS, the name of an OMS Output Device may be used as the first argument to specify the port.

int — A number alone can be used in place of a letter and number combination. Numbers 1-16 specify a channel number on port a, numbers 17-32 specify channels 1-16 on port b, etc. In this case, channel numbers received in the right inlet will specify both the port and the channel number.

If the MIDI Manager is being used, a letter a-f specifies a virtual port. If the MIDI Manager is not being used, the letter a specifies the modem port and b specifies the printer port. If OMS is being used, the letter a-z can be an abbreviation for any OMS or MIDI Manager port (configured using the **MIDI Setup...** dialog).

polyout

Transmit MIDI poly pressure messages

Output

(MIDI) There are no outlets. The output is a MIDI polyphonic key pressure message transmitted directly to the serial ports, or to the *virtual ports* of the MIDI Manager and/or OMS.

Examples

Letter argument transmits to only one port.
Otherwise, number specifies both port and channel

See Also

polyin	Ouput incoming MIDI polyphonic key pressure values
midiout	Transmit raw MIDI data
OMS	Using Max with OMS
Ports	How ports are specified
Tutorial 16	More MIDI ins and outs

Put one message at the beginning of another

prepend

Input

set — The word set, followed by any message, will replace the message stored in **prepend**, without triggering output.

anything else — The message stored in **prepend** is attached to the beginning of the message received in the inlet, and the combined message is sent out its outlet.

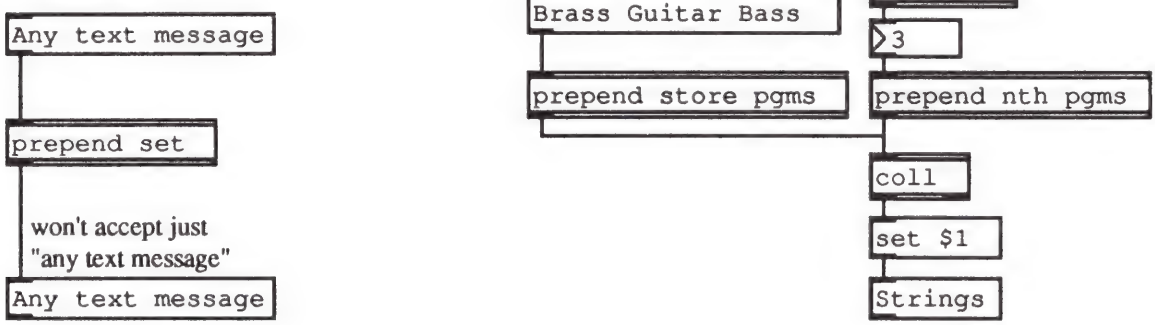
Arguments

anything — Obligatory. Sets the message to be prepended at the beginning of incoming messages. The first argument must be a symbol.

Output

anything — The message received in the inlet is combined with the message stored in **prepend**, and then sent out the outlet.

Examples



Symbols can be combined into meaningful messages with **prepend**

See Also

append — Append arguments at the end of a message
message — Output any message
Tutorial 25 — Managing messages

preset

Store and recall the settings of other objects

Input

store — When **preset** receives the word store, followed by a number, it stores the current setting of all user interface objects in the same window, in the preset indicated by the number. If objects are connected to **preset**'s outlet with patch cords, only the connected objects will be affected.

The presets (storage locations in the **preset** object) are numbered left-to-right, top-to-bottom. When settings are stored in a preset, a dot appears on it to indicate that it contains something. Settings can also be stored in a preset by holding down the Shift key and clicking on the preset with the mouse.

int — The number indicates a preset, and the settings stored in that preset are sent out to the connected objects, or to all objects in the window if no patch cords are connected to **preset**'s outlet. The settings in a preset can also be sent out by clicking on the preset with the mouse.

clear — Erases the contents of the most recently sent preset. The word clear, followed by a number, erases the contents of that numbered preset.

clearall — Erases the contents of all presets.

write — Calls up the standard **Save As...** dialog box, so that presets can be saved in a file. The word write, followed by a number, will save only that number preset in a file.

read — Calls up the standard **Open...** dialog box, so that presets can be read in from another file.

Arguments

(Get Info...) — A **preset** object can be set to save its contents as part of the patch that contains it. When the Patcher window is in Edit mode, select the **preset** object, choose **Get Info...** from the Max menu, and check the box next to *Save Presets with Patcher*. The **Get Info...** dialog box also allows you to assign a name to the **preset** object, to distinguish it from other **preset** objects in the same patch.

Output

int or float — When a preset is recalled, either by a mouse click or by a number in the inlet, the settings stored in that preset are sent out the outlet to all connected objects, or, if no objects are connected, to all user interface objects in the window.

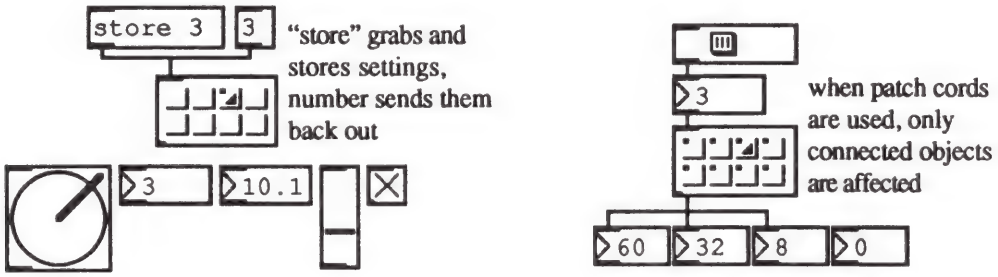
The objects which can be affected by **preset** are: **dial**, **Ggate**, **Gswitch**, **hslider**, **Number box** (both int and float), **slider**, **toggle**, and **uslider**. The contents of a **table** can also be stored and recalled by **preset**, but the **table** *must* be connected to **preset**'s outlet with a patch cord. The outlet of **preset** can also be connected to a **send** object, to communicate with objects connected to a **receive** object of the same name.

Store and recall the settings of other objects

preset

The number of visible presets can be adjusted by resizing the **preset**'s box. The maximum number of presets in a single **preset** object is 256.

Examples



Remember many past settings and recall them later

See Also

Data Structures	Ways of storing data in Max
grab	Intercept the output of another output
Tutorial 37	Data structures

print

Print any message in the Max window

Input

anything — Messages are not interpreted by the **print** object. They are simply printed verbatim in the Max window.

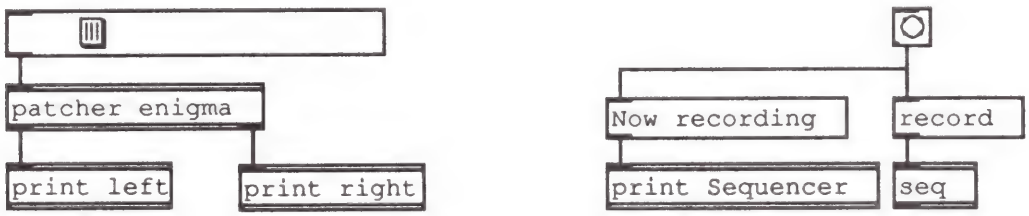
Arguments

anything — Optional. The argument is an identifier for the **print** object. Each message printed in the Max window is preceded by the name of the **print** object, and a colon (:). The name must not contain spaces or special characters, but can be either a number or a word. If there is no argument, the name of the **print** object is print. Using an argument to **print** can help distinguish the output of two or more **print** objects.

Output

anything — There are no outlets. The message received in the inlet is printed in the Max window.

Examples



Used for displaying output, or for notifying when an event takes place

See Also

Debugging	Techniques for debugging patches
Tutorial 1	Saying "Hello!"

Generate a random number

random

Input

bang — In left inlet: Sends out a randomly generated number between 0 and one less than its maximum limit.

int — In right inlet: The number is stored as the maximum limit for the random output. The output will always be between 0 and one less than this maximum limit.

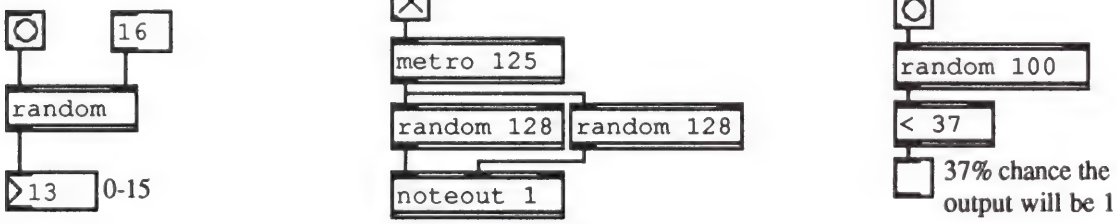
Arguments

int — Optional. Sets an initial limit to the random output. The output will always be between 0 and one less than this maximum limit. If there is no argument, the limit is initially set to 1, which causes **random** to output 0 whenever it receives a bang.

Output

int — When a bang is received in the left inlet, **random** generates a random number between 0 and one less than its maximum limit.

Examples



Generate random events, or make decisions based on probability

See Also

drunk — Output random numbers in a moving range
Tutorial 22 — Delay lines

receive / r

Receive messages, without patch cords

Input

anything — There are no inlets. Input is received from **send** objects that have the same name, even if the **send** is in another loaded patch.

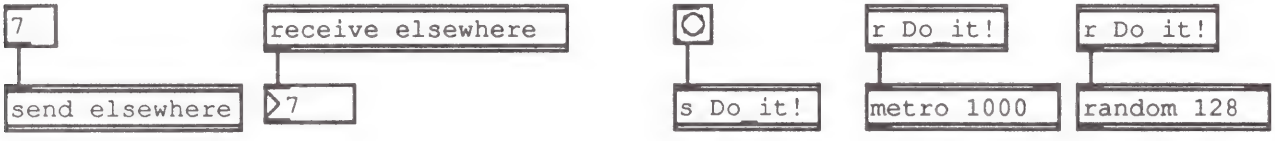
Arguments

any symbol — Obligatory. Gives a name to **receive**.

Output

anything — Any message received in the inlet of any **send** with the same name is sent out the outlet of **receive**, even if the **send** is in another loaded patch.

Examples



Virtual connections exist between all **send** *and* **receive** *objects which share the same name*

See Also

send	Send messages, without patch cords
value	Share a stored number with other objects
Tutorial 24	**send** and **receive**

Draw solid rectangle in graphics window

rect

Input

bang — In left inlet: Draws the rectangle using the current screen coordinates, drawing mode, and color.

int — In left inlet: Sets the left screen coordinate of the rectangle and draws the shape.

In 2nd inlet: Sets the top screen coordinate of the rectangle.

In 3rd inlet: Sets the right screen coordinate of the rectangle.

In 4th inlet: Sets the bottom screen coordinate of the rectangle.

In 5th inlet: Sets the drawing mode of the rectangle. See the listing of drawing mode constants under **oval**.

In 6th (right) inlet: Sets the palette index (color) of the rectangle according to the graphics window's current palette. This setting has no effect when the monitor is in black and white mode.

priority — In left inlet: The word priority, followed by a number greater than 0, sets a **rect** object's sprite priority in its graphics window. Objects with lower priority will draw behind those with a higher priority.

Arguments

any symbol — Obligatory. The first argument to **rect** must be the name of a graphics window into which the rectangle will be drawn. The window need not exist at the time the **rect** object is created, but the rectangle will not be drawn unless the name matches that of an existing and visible window.

int — Optional. Sets the initial sprite priority of the **rect**. If no priority is specified, the default is 3.

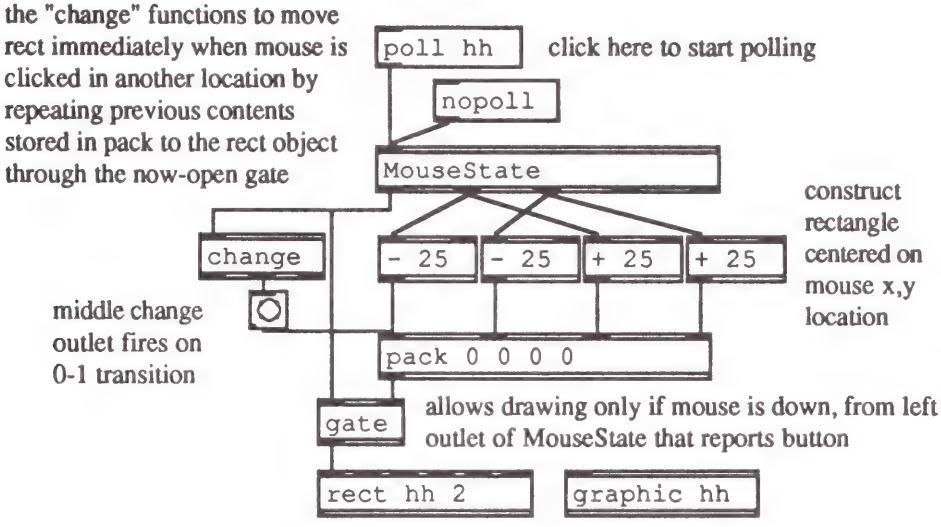
Output

(visual) — When the **rect** object's associated graphics window is visible, and a bang message or number is received in its left inlet, a shape is drawn in the window, and the object's previously drawn rectangle (if any) is erased.

rect

Draw solid rectangle in graphics window

Examples


the "change" functions to move rect immediately when mouse is clicked in another location by repeating previous contents stored in pack to the rect object through the now-open gate
poll hh
click here to start polling
nopoll
MouseState
construct rectangle centered on mouse x,y location
change
- 25
- 25
+ 25
+ 25
middle change outlet fires on 0-1 transition
pack 0 0 0 0
gate
allows drawing only if mouse is down, from left outlet of MouseState that reports button
rect hh 2
graphic hh

The **rect** *object in a paint program application*

See Also

Graphics	Overview of Max graphics windows and objects
graphic	Graphics window object
frame	Draw framed rectangle in graphics window
oval	Draw solid oval in graphics window
ring	Draw framed oval in graphics window

Draw framed oval in graphics window

ring

nput

bang — In left inlet: Draws a framed oval using the current screen coordinates, drawing mode, and color.

int — In left inlet: Sets the left screen coordinate of the oval and draws the shape.

In 2nd inlet: Sets the top screen coordinate of the oval.

In 3rd inlet: Sets the right screen coordinate of the oval.

In 4th inlet: Sets the bottom screen coordinate of the oval.

In 5th inlet: Sets the drawing mode of the oval. See the listing of drawing mode constants under **oval**.

In 6th (right) inlet: Sets the palette index (color) of the oval according to the graphics window's current palette. This setting has no effect when the monitor is in black and white mode.

priority — In left inlet: The word priority, followed by a number greater than 0, sets a **ring** object's sprite priority in its graphics window. Objects with lower priority will draw behind those with a higher priority.

Arguments

any symbol — Obligatory. The first argument to **ring** must be the name of a graphics window into which the oval will be drawn. The window need not exist at the time the **ring** object is created, but the oval will not be drawn unless the name matches that of an existing and visible window.

int — Optional. Sets the initial sprite priority of the **ring**. If no priority is specified, the default is 3.

Output

(visual) — When the **ring** object's associated graphics window is visible, and a bang message or number is received in its left inlet, a shape is drawn in the window, and the object's previously drawn oval (if any) is erased.

Examples

See examples under **oval** or **rect**. **ring** can be directly substituted for **oval**, **rect**, or **frame**.

ring

Draw framed oval in graphics window

See Also

Graphics	Overview of Max graphics windows and objects
graphic	Graphics window object
frame	Draw framed rectangle in graphics window
oval	Draw solid oval in graphics window
rect	Draw solid rectangle in graphics window

Selectively pass the input out a specific outlet

route

Input

anything — If the first item of the message is the same as one of the arguments of **route**, the rest of the message is sent out the outlet that corresponds to that argument. If the first item does not match any of the arguments, the entire message is passed out the rightmost outlet.

Arguments

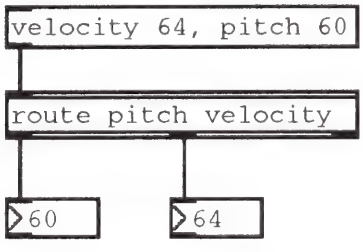
int, symbol — Optional. Arguments must be either all ints or all symbols. The number of arguments determines the number of outlets, *in addition to* the rightmost outlet. Each argument assigns a name or a number to an outlet. The maximum number of arguments is 10. If there is no argument, there is one other outlet, which is assigned the number 0.

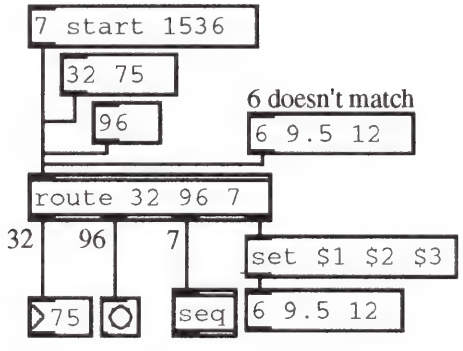
Output

anything — The first item of any message received in the inlet is compared with the arguments. If it matches one of the arguments, the rest of the message is sent out the specified outlet. Otherwise, the entire message is passed out the rightmost outlet.

If the first item of a message matches one of the arguments, but the message has no additional items, bang is sent out the specified outlet.

Examples





Arguments assign names or numbers to the outlets, and route the input to the correct outlet

See Also

gate — Pass the input out a specific outlet
select — Select certain inputs, pass the rest on
Tutorial 17 — Gates and switches

rtin

Output incoming MIDI real time messages

Input

(MIDI) rtin receives MIDI real time messages received directly from the serial ports, or from the *virtual ports* of the MIDI Manager and/or OMS.

port The word port, followed by a letter a-z or the name of an OMS Input Device, sets the port from which the object receives incoming MIDI messages. The word port is optional and may be omitted.

Arguments

a-z Optional. Specifies the port from which to receive incoming MIDI real time messages. If there is no argument, **rtin** receives from port a.

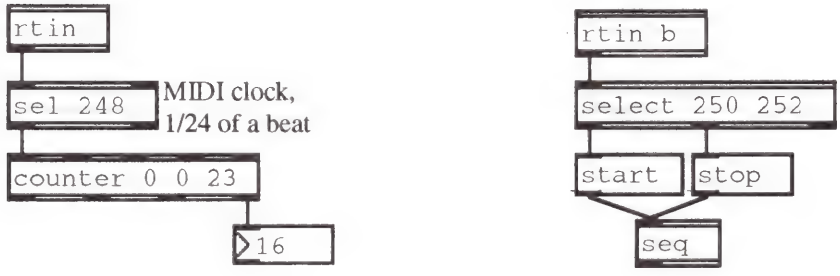
If the MIDI Manager is being used, a letter a-f specifies a virtual port. If the MIDI Manager is not being used, the letter a specifies the modem port and b specifies the printer port. If OMS is being used, the letter a-z can be an abbreviation for any OMS or MIDI Manager port (configured using the **Midi Setup...** dialog).

Note that if Filter Time Code is checked in the Apple MIDI Driver Settings window of the *PatchBay* application, MIDI clock messages will not be passed through to MIDI Manager applications. You may wish to use **timein** or an externally sync-ed **metro** or **clocker** in this case.

Output

int MIDI real time messages (MIDI clock, start, stop, and continue) received from the specified port are sent out the outlet.

Examples



MIDI real time messages can be used to synchronize Max with external events

See Also

clocker	Report elapsed time at regular intervals
MIDI	MIDI overview and specification
metro	Output bang at regular intervals
midiin	Receive raw MIDI data
OMS	Using Max with OMS
timein	Report time from external time code source
seq	Sequencer for recording and playing MIDI
Tutorial 16	More MIDI ins and outs

This page intentionally left blank

Select certain inputs, pass the rest on

select / sel

Input

int or float — In left inlet: If the number matches one of the arguments, a bang is sent out the outlet that corresponds to that argument. Otherwise, the number is passed out the rightmost outlet.

int — In right inlet: Replaces the value of the argument. The right inlet exists only if there is a single int argument.

any symbol — In left inlet: If the symbol is the same as one of the arguments, a bang is sent out the outlet that corresponds to that argument. Otherwise, the symbol is passed out the rightmost outlet.

Arguments

anything — Optional. The arguments can be ints, floats, or symbols. The number of arguments determines the number of outlets in addition to the rightmost outlet. If there is no argument, there is only one other outlet, which is assigned the number 0. The maximum number of arguments is 10.

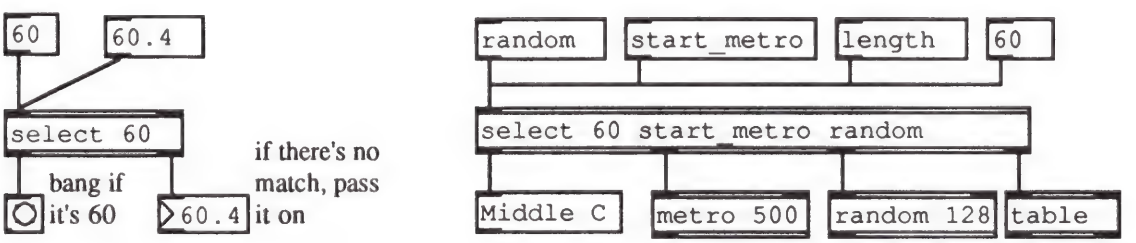
int — If there is a single int argument (or if there are *no* arguments) a second inlet is created on the right. Numbers received in that inlet are stored in place of the argument. If there is more than one argument, or if only argument is not an int, the right inlet is not created.

Output

bang — If the number or symbol received in the left inlet is the same as one of the arguments, a bang is sent out the outlet that corresponds to that argument.

anything — If the number or symbol received in the left inlet does not match any of the arguments, it is passed out the rightmost outlet.

Examples



Arguments assign names or numbers to the outlets, and a bang *is sent when the input matches them*

select / sel

Select certain inputs, pass the rest on

See Also

route	Selectively pass the input out a specific outlet
==	Compare two numbers, ouput 1 if they are equal
Tutorial 17	Gates and switches

Send messages,
without patch cords

send / s

Input

anything — A message received in the inlet is sent out the outlet of any **receive** object that has the same name, even if the **receive** is in another loaded patch.

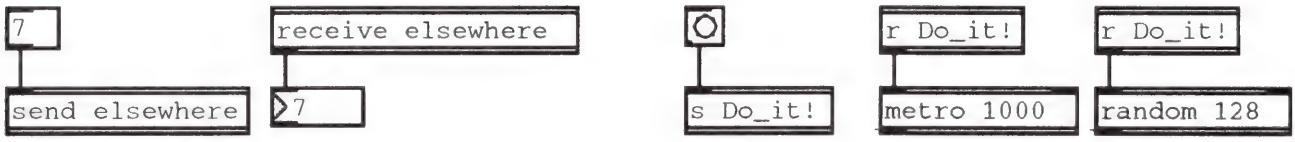
Arguments

any symbol — Obligatory. Gives a name to the **send** object.

Output

anything — There are no outlets. A message received in the inlet of **send** is sent out the outlet of any **receive** with the same name, even if the **receive** is in another loaded patch.

Examples



Virtual connections exist between all **send** *and* **receive** *objects which share the same name*

See Also

receive	Receive messages, without patch cords
value	Share a stored number with other objects
Tutorial 24	**send** and **receive**

serial

Send and receive characters from serial ports and cards

Input

int — Sends the number out the serial port accessed by the **serial** object. Numbers outside the range 0-255 are not sent.

list — Sends each number in the list out the serial port, from left to right. Numbers outside the range 0-255 are not sent.

bang — Sends, out the **serial** object's outlet, each character (as an integer) received on the serial port since the last bang message was received, in the order characters were received.

Arguments

symbol a-f — Obligatory. Specifies the serial port to be used by the object. a specifies the modem port, b the printer port, and c-f are ports on an installed serial card. See the **serial** help file for cards currently supported. If the port is already in use by MIDI or AppleTalk, or doesn't exist, an error message is displayed and the object will not function.

int — Optional. Specifies the baud rate of the serial port. Default value is 4800 baud. Other allowable rates are 300, 600, 1200, 1800, 2400, 3600, 7200, 9600, 19200, 38400, and 57600.

int — Optional. Specifies the number of data bits for the serial port. Default is 8 data bits. Other possible values are 5, 6, and 7.

int — Optional. Specifies the number of stop bits for the serial port. Default is 1. Other possible values are 1.5 and 2.

symbol — Optional. Specifies the parity for the serial port. Default is no parity (no), other possible values are even and odd.

Output

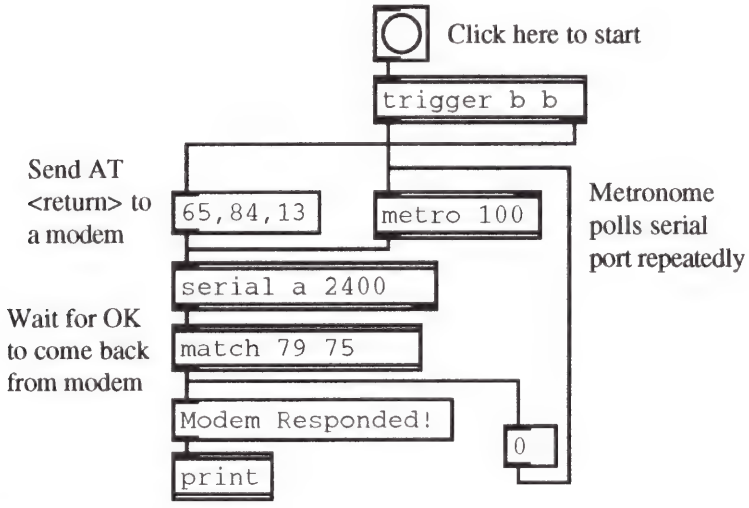
(serial output) — When a number or list is received in its inlet, **serial** sends the data out the specified serial port at the current baud rate.

int — When **serial** receives a bang message and characters have been received in the serial port, the received characters are sent as numbers in the order they were received.

Send and receive characters from serial ports and cards

serial

Examples



When upper button is clicked, this patch gets a modem's attention, begins polling for a response, and stops polling when a response has been received

See Also

match	Look for a series of numbers, output it as a list
vdp	Control a videodisc player through the serial port

seq

Sequencer for recording and playing MIDI

Input

bang — Starts playing the sequence stored in **seq**.

start — The word start by itself has the same effect as bang. The word start, followed by a number, plays the stored sequence at a tempo determined by the number. The message start 1024 indicates normal tempo. If the number is 512, **seq** plays the sequence at half the original recorded speed, start 2048 plays it back at twice the original speed, and so on.

record — Starts recording MIDI messages received in the inlet.

stop — Stops the sequencer if it is recording or playing. A stop message need not be received when switching directly from playing to recording, or vice-versa.

append — Starts recording at the end of the stored sequence, without erasing the existing sequence.

int — When **seq** is recording, numbers received in its inlet are interpreted as bytes of MIDI messages (usually from **midiformat** or **midiin**). MIDI channel messages and system exclusive messages can be recorded by **seq**, but **seq** does not respond directly to MIDI real time messages such as start, stop, MIDI clock, etc.

float — Converted to int.

delay — The word delay, followed by a number, sets the onset time, in milliseconds, of the first event in the recorded sequence.

hook — The word hook, followed by a float, multiplies all the event times in the stored sequence by that number. For example, if the number is 2.0, all event times will be doubled, and the sequence will play back twice as slowly. Multiplications can even be performed while the sequence is playing.

write — Calls up the standard **Save As...** dialog box, so that a recorded sequence can be saved as a separate file. If you want to edit the sequence with the text editor, check the *Save As Text* option in the dialog box.

read — With no arguments, read calls up the standard **Open...** dialog box, so that a previously recorded sequence can be read into **seq**, replacing the current sequence. With a symbol as an argument, read searches for a file with the specified name to read into the **seq** object.

print — Prints the first few events of the recorded sequence in the Max window.

Arguments

any symbol — Optional. Specifies the name of a file to be read into **seq** automatically when the patch is loaded.

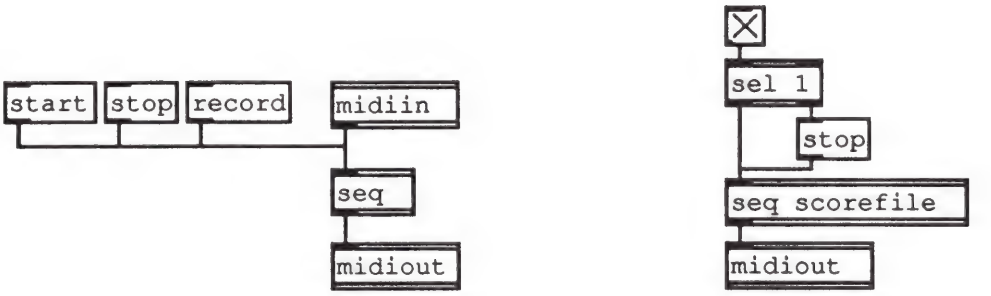
Sequencer for recording and playing MIDI

seq

Output

int When bang or start is received in the inlet, the sequence stored in **seq** is sent out the outlet in the form of individual MIDI bytes, usually to be sent to **midiparse** or **midiout**.

Examples



Record and play back live performance, or play a pre-recorded sequence

See Also

follow	Compare a live performance to a recorded performance
mtr	Multi-track sequencer
Sequencing	Recording and playing performances with MIDI
Tutorial 35	**seq** and **follow**

slider

Output numbers by moving a slider

Input

int — The number received in the inlet is displayed graphically by **slider**, and is passed out the outlet. Optionally, **slider** can multiply the number by some amount and add an offset to it, before sending it out the outlet.

The **slider** will also send out numbers in response to dragging on it directly with the mouse.

float — Converted to int.

bang — Sends out the number currently stored in the **slider**.

set — The word set, followed by a number, resets the value displayed by the **slider**, without triggering output.

size — The word size, followed by a number, resets the range of the **slider**. Any number less than 1 will be set to 1.

Arguments

The range of **slider** is set by selecting it (when the Patcher window is unlocked) and choosing **Get Info...** from the Max menu. The **slider** automatically resizes itself to accomodate the new range.

The **Get Info...** dialog box also provides a *Multiplier*—by which all numbers will be multiplied before being sent out, and an *Offset*—which will be added to the number, after multiplication. A newly created **slider** has a range of 128, a multiplier of 1, and an offset of 0.

Output

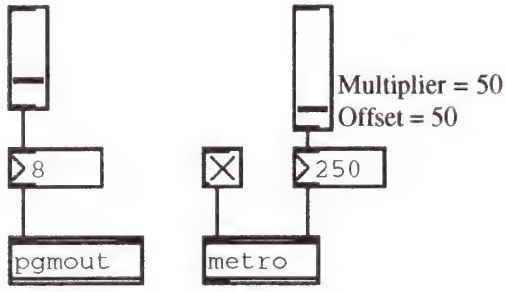
int — Numbers received in the inlet, or produced by dragging on **slider** with the mouse, are first multiplied by the multiplier, then have the offset added to them, then are sent out the outlet.

Although the numbers that can be output by dragging are limited by the range of the **slider**, numbers received in the inlet are not limited before they are sent out the outlet.

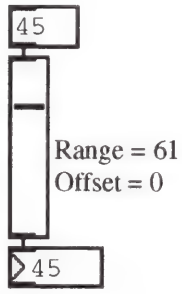
Output numbers by moving a slider

slider

Examples



Produce output by dragging onscreen...



or display numbers passing through

See Also

slider exists for compatibility with old Max files. **hslider** and **uslider** have many more features.

dial	Output numbers by moving a dial onscreen
hslider, uslider	Output numbers by moving a slider onscreen
Tutorial 9	Using the slider
Tutorial 14	Sliders and dials

snd

Play a sound through internal speaker

Input

int — In left inlet: If non-zero, plays the current sound through the Macintosh speaker. If zero, stops any currently playing sound.

bang — Same as non-zero int, plays the sound.

stop — Same as 0, stops the sound.

read — With no arguments, read calls up the standard **Open...** dialog box, so that a file containing 'snd ' resources can be read into the **snd** object. Note that the dialog box will show all types of files, since there is no standard file type for sound resources. If you choose a file which does not contain sound resources, you will see the error message "snd: no 'snd ' resources" in the Max window. With a symbol as an argument, read searches for a file with the specified name to read into the **snd** object. Note that you must send a load message after reading in a new sound in order for a sound in the file to be heard.

load — With no arguments, load disposes of the previous sampled sound (if any) and loads the first (or only) 'snd ' resource into memory. With a symbol as an argument, load tries to load an 'snd ' resource of that name.

dispose — Frees the "sound channel" occupied by the **snd** object. This is important if you wish to hear other sounds through the Macintosh speaker. 0 has the same effect, but unless you stop a sound explicitly or send the dispose message, the **snd** object will occupy the channel. However, this channel *can* be used by another **snd** object.

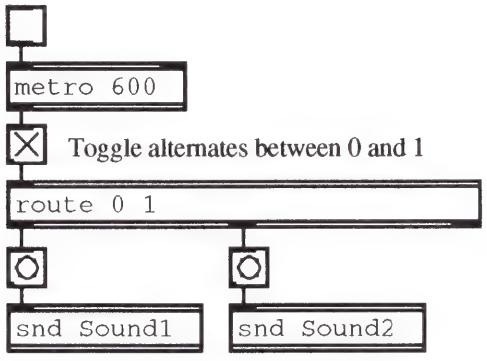
Arguments

any symbol — Optional. Specifies the name of a sound resource file to be read into **snd** automatically when the patch is loaded.

Output

(sound) — When a non-zero number is received in its inlet, **snd** plays its current sound through the speaker.

Examples



Alternate between two different sounds

speedlim

Limit the speed that numbers can pass through

Input

int — In left inlet: The number will be passed out the outlet, provided that a certain minimum time has elapsed since the previous output. Otherwise, the number is ignored.

In right inlet: The number is stored as the minimum amount of time, in milliseconds, between successive outputs.

float — Converted to int.

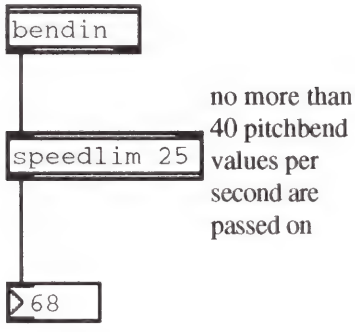
Arguments

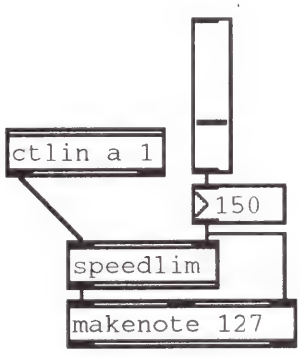
int — Optional. Sets an initial minimum time between outputs, in milliseconds. If there is no argument, the minimum time is 0.

Output

int — A number received in the left inlet is sent out the outlet, provided the specified minimum amount of time has elapsed since the previous output. Otherwise, the number is ignored.

Examples





Used to reduce a heavy flow of numbers, or to turn a continuous flow into discrete steps

See Also

mousefilter	Pass numbers only when the mouse button is up
timer	Report elapsed time between two events
Tutorial 16	More MIDI ins and outs

Look for a range of numbers

split

Input

int — In left inlet: If the number is within a specified range, it is sent out the left outlet. Otherwise, it is sent out the right outlet.

In middle inlet: The number is stored as the minimum value in the range of numbers looked for by **split**.

In right inlet: The number is stored as the maximum value in the range of numbers looked for by **split**.

list — In left inlet: The second number is stored as the minimum value of the range, and the third number is stored as the maximum value of the range. The first number is then compared to the range, and is sent out one of the two outlets.

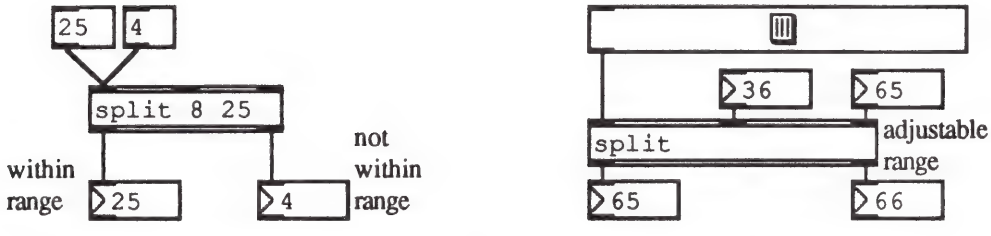
Arguments

int — Optional. The first argument sets the minimum value to be sent out the left outlet. The second argument sets the maximum value to be sent out the left outlet.

Output

int — If the number received in the left inlet is *greater than or equal to* the specified minimum, *and* it is *less than or equal to* the specified maximum, it is sent out the left outlet. Otherwise, it is sent out the right outlet.

Examples



Used to divert a certain range of numbers to a different destination

See Also

select	Select certain inputs, pass the rest on
<=	*Is less than or equal to,* comparison of two numbers
>=	*Is greater than or equal to,* comparison of two numbers
Tutorial 20	Using the Macintosh keyboard

stripnote

Filter out note-off messages, pass only note-on messages

Input

list In left inlet: The second number is stored as a velocity, and the first number is treated as the pitch, of a MIDI note-on message. If the second number is not 0, it is sent out the right outlet, and the first number is sent out the left outlet. If the second number is 0, nothing is sent out.

int In left inlet: The number is treated as a pitch value. If the velocity value currently held by **stripnote** is not 0, then the velocity is sent out the right outlet and the pitch is sent out the left outlet.

In right inlet: The number is stored as a velocity to be paired with pitch numbers received in the left inlet.

Arguments

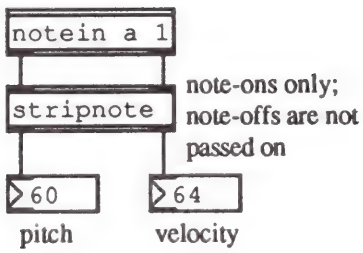
None.

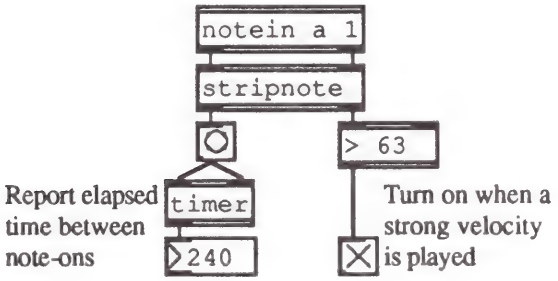
Output

int Out left outlet: The pitch value received in the left inlet is sent out, provided the velocity is not 0.

Out right outlet: The velocity value of a note-on pair is sent out, provided it is not 0.

Examples





Repeated pitch values and 0 velocities caused by note-off messages can be filtered out

See Also

makenote	Generate a note-off message, following each note-on
sustain	Hold note-off messages, output them on command
Tutorial 13	Managing note data

Hold note-off messages, output them on command

sustain

Input

list In left inlet: The second number is stored as the velocity, and the first number is treated as the pitch, of a MIDI note-on message. If the pair is a note-on (the velocity is not 0), the velocity is sent out the right outlet and the pitch is sent out the left outlet. Note-offs (note-ons with a velocity of 0) are either passed on immediately or held by **sustain**.

int In left inlet: The number is the pitch value of a pitch-velocity pair. If the velocity value currently held by **sustain** is not 0, then the pair is sent out immediately. If the velocity is 0, the note-off is either sent out or held, depending on whether **sustain** is turned on.

In middle inlet: The number is stored as a velocity to be paired with pitch numbers received in the left inlet.

In right inlet: If the number is not 0, **sustain** is turned on, and all note-offs are held. If the number is 0, **sustain** is turned off, and all note-offs are sent out immediately.

Arguments

None.

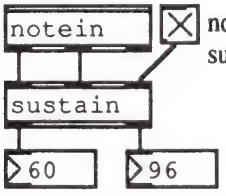
Output

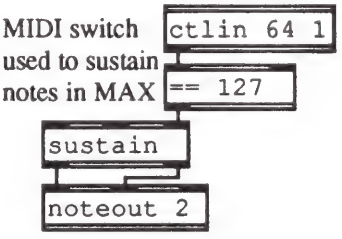
int Out left outlet: The pitch value of a pitch-velocity pair.

Out right outlet: The velocity value of a pitch-velocity pair.

Note-on pairs are always sent out immediately. If **sustain** is turned on, *note-offs* are held until it is turned off. Otherwise, note-offs are sent out immediately.

Examples





Like the sustain pedal of a piano, **sustain** *releases all held notes at one time*

sustain

Hold note-off messages, output them on command

See Also

flush	Provide note-offs for held notes
makenote	Generate a note-off message, following each note-on
stripnote	Filter out note-off messages, pass only note-on messages

Reverse the sequential order of two numbers

swap

Input

int — In left inlet: The number is sent out the right outlet, then the number in the right inlet is sent out the left outlet.

In right inlet: The number is stored to be sent out the left outlet when a number is received in the left inlet.

float — The numbers are converted to int, unless there is a float argument, in which case the number received in the right inlet is stored as a float.

list — In left inlet: The numbers are stored in **swap**. The first number is sent out the right outlet, then the second number is sent out the left outlet.

bang — In left inlet: Swaps and sends out the numbers currently stored in **swap**.

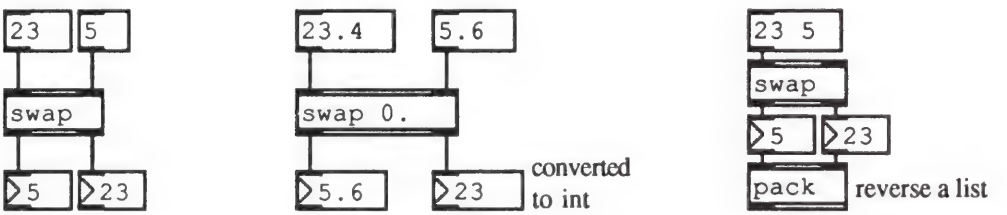
Arguments

int or float — Optional. Sets an initial value for the number which is to be sent out the left outlet. Argument with a decimal point will cause a float to be sent out the left outlet. (The number sent out the right outlet is *always* an int.) If there is no argument, the initial value is 0.

Output

int — When a number is received in the left inlet, the number in each inlet is sent out the opposite outlet.

Examples



Numbers are sent out in reverse order from that in which they were received

See Also

fswap	Reverse the sequential order of two decimal numbers
pack	Combine numbers into a list
unpack	Break a list up into individual numbers
Tutorial 30	Number groups

switch

Receive the input in a specific inlet

Input

int — In left inlet: The number specifies an open inlet for receiving subsequent messages sending them out the outlet. All inlets other than the designated open one are closed. If the number is 0, all inlets are closed.

anything — In any other inlet: Any message received in an *open* inlet is passed out the outlet. Messages received in closed inlets are ignored.

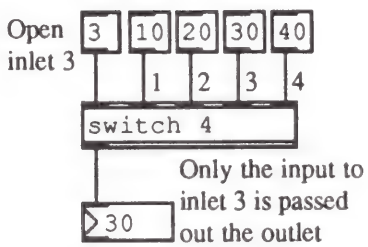
Arguments

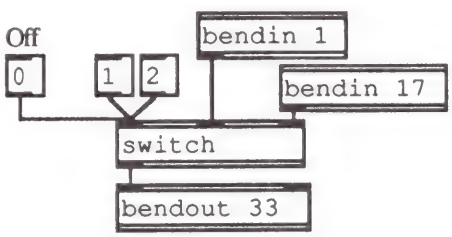
int — Optional. Specifies the number of inlets, up to 10, *in addition to* the leftmost inlet. If there is no argument, there are two additional inlets.

Output

anything — Messages received in the open inlet (the one specified by the number in the left inlet) are passed out the outlet. If the number in the left inlet is 0, or if no inlet number has been received, all messages are ignored. If the number in the left inlet is less than 0, messages are received in the inlet just to the right of the leftmost inlet. If the number is greater than the number of existing inlets, messages are received in the rightmost inlet.

Examples





"Listen" to only one inlet at a time, or ignore all inlets

See Also

gate — Pass the input out a specific outlet
Gswitch — Receive the input in one of two inlets
Tutorial 17 — Gates and switches

Prepare MIDI system exclusive messages

sxformat

Input

int In left inlet: The number replaces any $i1 arguments in the object box, and the entire list of arguments is evaluated and sent out the outlet, one-by-one.

In other inlets: The number is stored in place of the $i argument that corresponds to that inlet, until a number is received in the left inlet.

list In left inlet: The numbers in the list are used to replace the corresponding $i arguments in the object box, then the list of arguments is evaluated and the numbers are sent out one-by-one.

Arguments

list Obligatory. The arguments are a list of numbers which represent the values of individual bytes of a MIDI system exclusive message. The first number should be 240 (or 0xF0), the system exclusive status byte, and the last number should be 247 (or 0xF7), the end byte. There can be any number of values for data bytes in between.

Arguments for data bytes can also be in the form of a mathematical expression (like the expressions in **expr** and **if** objects) to be evaluated before numbers are sent out the outlet. The expressions can contain changeable arguments in the form $i, followed immediately by an inlet number (for example, $i2). The changeable arguments are replaced by numbers received in the specified inlet. Expressions used in place of numbers should be preceded by the word is, and should be separated from other arguments with a slash (/) on either side of the expression (see example).

If the value of an evaluated expression is less than 0, no number is sent out in place of that expression. This allows you to send variable-length system exclusive messages.

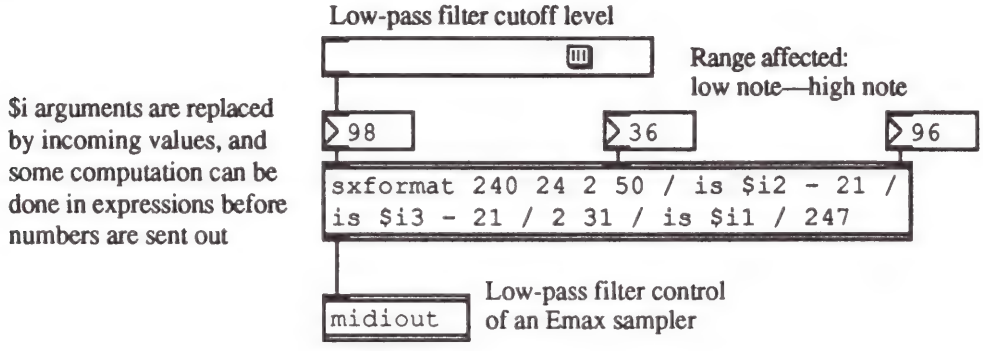
Output

int When a number is received in the left inlet, any expressions in the argument are evaluated and the numbers in the list are sent out one at a time, as bytes of a MIDI system exclusive message, for transmission by **midiout**.

sxformat

Prepare MIDI system exclusive messages

Examples


Low-pass filter cutoff level
Range affected:
low note—high note
$i arguments are replaced by incoming values, and some computation can be done in expressions before numbers are sent out
98
36
96
sxformat 240 24 2 50 / is $i2 - 21 /
is $i3 - 21 / 2 31 / is $i1 / 247
midiout
Low-pass filter control of an Emax sampler

sxformat *can send a complete MIDI system exclusive message, byte-by-byte, to* **midiout**

See Also

expr	Evaluate a mathematical expression
midiout	Transmit raw MIDI data
sysexin	Output incoming MIDI system exclusive messages
Tutorial 34	Managing raw MIDI data

Output incoming MIDI system exclusive messages

sysexin

Input

(MIDI) sysexin receives MIDI system exclusive messages directly from the serial ports, or from the *virtual ports* of the MIDI Manager and/or OMS.

port The word port, followed by a letter a-z or the name of an OMS Input Device, sets the port from which the object receives incoming MIDI messages. The word port is optional and may be omitted.

Arguments

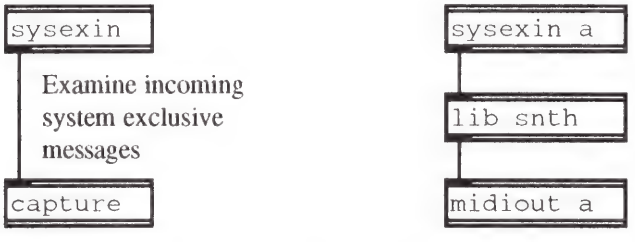
a-z Optional. Specifies the port from which to receive incoming MIDI system exclusive messages. If there is no argument, **sysexin** receives from port a.

If the MIDI Manager is being used, a letter a-f specifies a virtual port. If the MIDI Manager is not being used, the letter a specifies the modem port and b specifies the printer port. If OMS is being used, the letter a-z can be an abbreviation for any OMS or MIDI Manager port (configured using the **Midi Setup...** dialog).

Output

int MIDI system exclusive messages received from the specified port are sent out the outlet, byte-by-byte.

Examples



MIDI system exclusive messages can be interpreted by a **lib** *object*

See Also

MIDI	MIDI overview and specification
midiin	Receive raw MIDI data
lib	Script-driven patch librarian
Tutorial 34	Managing raw MIDI data

table

Store and graphically edit an array of numbers

Input

list — In left inlet: The second number is stored in **table**, at the address (index) specified by the first number.

int — In left inlet: The number specifies an address in the **table**. The value stored at that address is sent out the left outlet. *However*, if a value has been received in the right inlet, **table** *stores* that value in the specified address, rather than sending out a number.

In right inlet: The number specifies a value to be stored in **table**. The next address number received in the left inlet causes the value to be stored at that address.

float — Converted to int.

send — The word send, followed by the name of a **receive** object, followed by an address number, sends the value stored at that address to all **receive** objects with that name, without sending the value out the **table**'s outlet.

length — In left inlet: Sends the length (size) of the **table** out the left outlet.

size — In left inlet: The word size, followed by a number, sets the size of the **table** to that number.

set — In left inlet: The word set, followed by a list of numbers, stores values in certain addresses. The first number after the word set specifies an address. The next number is the value to be stored in that address, and each number after that is stored in a successive address.

clear — In left inlet: Sets all values in the **table** to 0.

read — In left inlet: The word read, followed by a name, opens and reads data values from a file in Text or Max binary format. Without an argument, read opens a standard open file dialog for choosing a file to read values from. If the file contains valid data, the entire contents of the existing table are replaced with the data.

write — In left inlet: Opens a standard save file dialog for choosing a name to write data values from the table. The file can be saved in Text or Max binary format.

goto — In left inlet: The word goto, followed by a number, sets a pointer to the address specified by the number. The pointer is set at the beginning of the **table** initially.

next — In left inlet: Sends the value stored in the address pointed at by the goto pointer out the left outlet, then sets the pointer to the next address. If the pointer is currently at the last address in the table, it *wraps around* to the first address.

prev — In left inlet: Causes the same output as the word next, but the pointer is then decremented rather than incremented. If the pointer is currently at the first address in the table, it *wraps around* to the last address.

Store and graphically edit an array of numbers

table

max — In left inlet: Sends the maximum stored value out the left outlet.

min — In left inlet: Sends the minimum stored value out the left outlet.

refer — In left inlet: The word refer, followed by the name of another **table**, sets the receiving **table** object to read its data values from the named **table**.

inv — In left inlet: The word inv, followed by a number, finds the first value which is greater than or equal to that number, and sends the *address* of that value out the left outlet.

sum — In left inlet: Sends the sum of all the values in the **table** out the left outlet.

quantile — In left inlet: The word quantile, followed by a number, multiplies the number by the sum of all the values in the **table**. This result is then divided by 2^{15} (32,768). Then, **table** sends out the address at which the sum of all values *up to* that address is greater than or equal to the result.

bang — In left inlet: Same as a quantile message with a random number between 0 and 32,768 as an argument.

(mouse) — The values stored in **table** can be entered and edited graphically with the mouse. When a **table** object is first created in a Patcher window, the **table**'s graphic editing window is opened, in which values can be entered by drawing with the mouse. The editing window provides a palette of graphic editing tools.

show crosshairs, to aid precision drawing

select a region to cut, copy, clear, or paste

drag the display, to see another part of the table

draw in values freehand with the mouse

draw values in a straight line, from click to click

zoom the horizontal display in or out

zoom the vertical display in or out

When the Patcher window is locked, the graphic editing window can be opened by double-clicking with the mouse on the **table** object .

A **table** can be created in a separate file by opening a new Table window and choosing the **Save** command from the File menu. A **table** can also be created in a separate file by opening a new Text file, and simply beginning the file with the word table. The word table should be followed by a list of space-separated numbers, specifying values to be stored in the **table**.

table

Store and graphically edit an array of numbers

A **table** which has been saved as a file can be viewed and edited as text by choosing **Open as Text...** from the File menu. Numbers in the form of text can be pasted in from other sources such as the editing window of a **capture** object, or even from another program such as a word processor. Text from a **capture** object can also be pasted directly into **table**'s graphic editing window.

(Get Info...) When the editing window is open, various traits of the **table** can be modified by choosing **Get Info...** from the Max menu.

Size determines the number of values stored in the **table**. A newly created **table** has 128 values, indexed with numbers from 0 to 127.

Range determines the range of values which can be displayed on the *y* axis of the editing window. A newly created **table** has a range of 128, from 0 to 127.

If *Signed* is checked, **table** displays negative numbers as well as positive. In effect, the range of displayed values specified by *Range* is doubled when the *Signed* option is checked, since the range goes in both directions from 0.

If *Note Name Legend* is checked, values are shown on the *y* axis as MIDI note names, rather than numbers.

If *Don't Save* is checked, Max will not ask if you want to save changes made to the **table**, when the patch containing that **table** is closed.

If *Save with Patcher* is checked, the values in the **table** are saved as part of the patch that contains it. Otherwise, the **table** has to be saved in a separate file to retain its values.

Arguments

any symbol Optional. The argument gives a name to the **table**. Max looks for a **table** of the same name which has been saved as a separate file. If two or more **table** objects share the same names, they also share the same data, even if Max couldn't find a file with the name.

Output

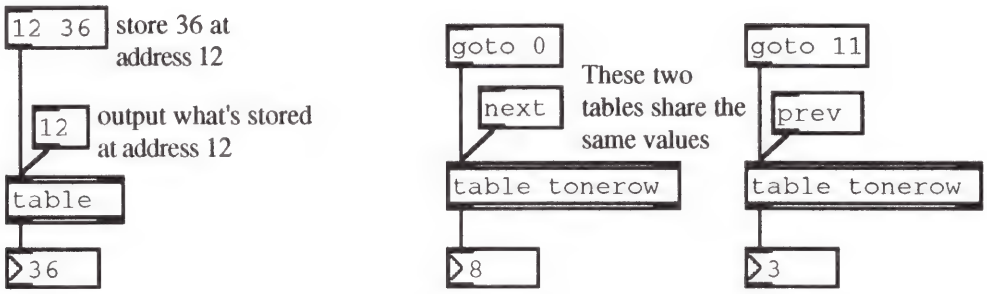
int All numbers sent out by **table** are sent out the left outlet.

bang When the contents of a table have been changed by an edit in the graphic editing window, bang is sent out the right outlet.

Store and graphically edit an array of numbers

table

Examples



An array of any size and range can be stored, recalled, and modified

See Also

capture	Store numbers to view or edit
coll	Store a collection of different messages
Data Structures	Ways of storing data in Max
funbuff	Store x,y pairs of numbers together
Histo	Make a histogram of the numbers received
Quantile	Using **table** for probability distribution
Tables	Using the **table** graphic editing window
Text	Format numbers as a text file
Tutorial 32	The **table** object

tempo

Output numbers at a metronomic tempo

Input

bang — In left inlet: Starts **tempo**'s metronome process, or *re*starts it if **tempo** is already on.

stop — In left inlet: Stops **tempo**.

int — In left inlet: If the number is not 0, it has the same effect as bang. If the number is 0, it has the same effect as stop.

In 2nd inlet: The number is stored as the tempo, in beats per minute (quarter notes per minute). The tempo is limited between 5 and 300 beats per minute.

In 3rd inlet: The number is a *beat multiplier*, which can lengthen the amount of time taken for one beat. It slows the tempo down by a factor. For example, a multiplier of 2 will make **tempo** send out its output half as fast.

In right inlet: The number is the rhythmic value sent out by **tempo**, specified as a fraction of a whole note. For example, the number 8 causes **tempo** to output eighth notes, relative to the specified (quarter note) tempo. The numbers sent out the outlet cycle continuously between 0 and the number 1 less than the rhythmic value. The divisions of a whole note must be between 1 and 96.

float — Converted to int.

tempo — In left inlet: The word tempo, followed by a float, sets the current tempo to the number, allowing tempos to be expressed in floating-point.

Arguments

int — Optional. The first argument sets an initial tempo, from 5 to 300 beats per minute. If there is no argument, the initial tempo is 120 beats per minute. The second argument is the beat multiplier and is always set to 1. The third argument sets an initial rhythmic value of the output, from a whole note (1) to a 64th note triplet (96). If the argument is not present, the initial value is 16.

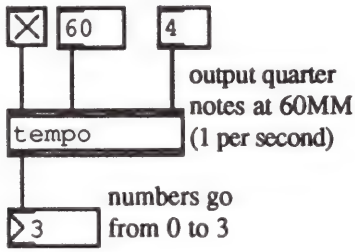
Output

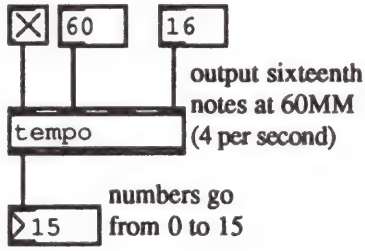
int — When **tempo** is started it outputs numbers in a continuous cycle from 0 to the number 1 less than the specified rhythmic value. The speed at which the numbers are sent out is determined by the tempo (quarter note beats per minute) and the rhythmic value of the output (fraction of a whole note).

Output numbers at a metronomic tempo

tempo

Examples





The tempo defines the speed of a quarter note, division defines the pulse to be sent out

See Also

clocker	Report elapsed time, at regular intervals
timein	Report time from external time code source
metro	Output bang, at regular intervals
Tutorial 31	Using timers

Text

Format numbers as a text file

Input

int or float — Numbers are stored in **Text** in the order in which they are received, separated by tab stops.

list — Only the first number in the list is stored.

tab — Puts an additional tab stop at the end of the stored numbers.

cr — Puts a carriage return at the end of the stored numbers, to start a new line.

clear — Erases the contents of **Text**.

write — Calls up the standard **Save As...** dialog box, so that the contents of **Text** can be saved in a separate Text file.

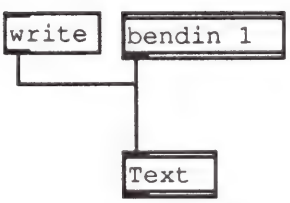
read — Calls up the standard **Open...** dialog box, so that a text file can be read in.

Double-clicking with the mouse on the **Text** object (when the Patcher window is locked) opens an editing window in which the contents of **Text** can be viewed and edited. No numbers can be received by **Text** while the editing window is open. Unlike the **capture** object, changes made in the editing window of **Text** actually alter the contents of the object.

Arguments

None.

Examples



Collect numbers to paste elsewhere, or to save as a separate file

See Also

capture — Store numbers to view or edit
table — Store and edit an array of numbers

Combine numbers into a list, when received close together

thresh

Input

int or float — In left inlet: Numbers are combined into a list if received within a certain time of each other. When the time between incoming numbers is greater than the specified threshold, the list is sent out the outlet, and a new list is started.

In right inlet: The number is stored as the time, in milliseconds, to wait before sending out the compiled list of numbers. If no new number is received in the left inlet within that time, the list is sent out and a new list is started.

list — In left inlet: The entire list is appended to the list stored in **thresh**.

Arguments

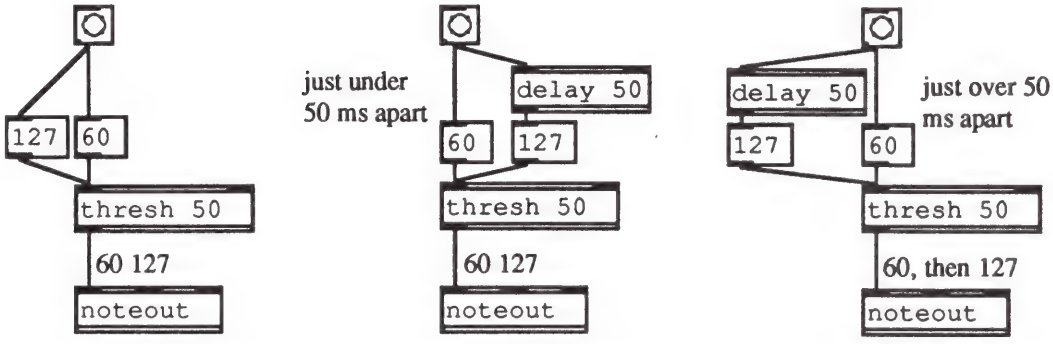
int — Optional. Sets an initial value for the threshold time. If no argument is present, the initial value is 10 milliseconds.

float — Converted to int.

Output

list — Each number received in the left inlet is appended to a list stored by **thresh**. If a certain time passes without a new number being received, **thresh** sends out the list and starts a new list.

Examples



If threshold time is exceeded without a new number being received, **thresh** *sends out what it holds*

See Also

iter — Break up a list into a series of numbers
pack — Combine numbers into a list
Tutorial 37 — Data structures

timein

Report time from external time code source

Input

int In left inlet: Any number other than 0 starts **timein**. At regular intervals, **timein** sends out the current time, as obtained from the MIDI Manager. The timing source can be an external device, another client application, or the MIDI Manager itself. 0 stops **timein**.

In middle inlet: The number specifies a time code format in which to report the current time. Time codes are numbered as follows:

0	milliseconds
1	beats
2	24 frames/sec bits
3	25 frames/sec bits
4	30 frames/sec Drop bits
5	30 frames/sec bits
6	24 frames/sec
7	25 frames/sec
8	30 frames/sec Drop
9	30 frames/sec.

In right inlet: The number is the time interval, in milliseconds, at which **timein** reports the current time. Any number less than 5 is set to 5. A new number in the right inlet does not take effect until the next output is sent.

float Converted to int.

Arguments

int Optional. The first argument sets an initial value for the time interval at which **timein** sends its output. If there is no argument, the initial time interval is 100 milliseconds. Any value less than 5 will be set to 20. The second argument specifies a time code format in which **timein** will display the time obtained from the MIDI Manager. If the argument is not present, it is set to milliseconds format.

Output

int Out left outlet: The current time is sent out at regular intervals (specified by the number in the rightmost inlet). For non-bits SMPTE formats (formats 6-9), time is reported in hours, minutes, seconds, and frames, so the left outlet reports hours.

Out 2nd oultet: When **timein** is reporting the time in a non-bits SMPTE format, the minutes are sent out this outlet.

Out 3rd oultet: When **timein** is reporting the time in a non-bits SMPTE format, the seconds are sent out this outlet.

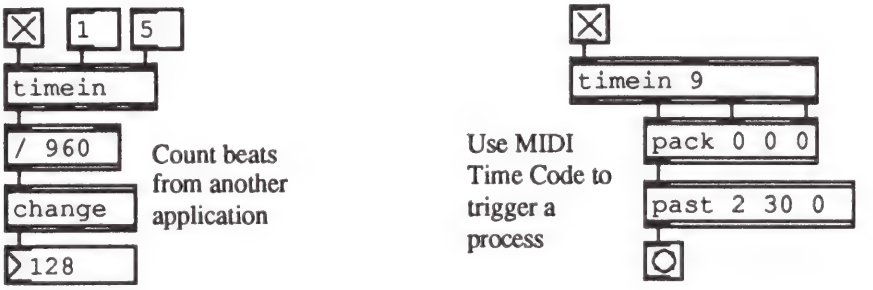
Report time from external time code source

timein

Out right outlet: When **timeIn** is reporting the time in a non-bits SMPTE format, the number of frames is sent out this outlet.

timeIn will not function unless Max is using the MIDI Manager.

Examples



The MIDI Manager lets you convert time from an external source into different time code formats

See Also

clocker	Report the elapsed time, at regular intervals
MIDI	MIDI overview and specification
Ports	How ports are specified
tempo	Output numbers at a metronomic tempo
timeout	Write current time to MIDI Manager time port

timeout

Write current time to MIDI Manager time port

Input

int — In left inlet: Sets the current time, then writes this time to Max's MIDI Manager Clock Out time port. When **timeout** is in non-bits SMPTE time format (6-9), the leftmost inlet sets the hours of the current time.

In 2nd inlet: Sets the minutes of the current time when **timeout** is in non-bits SMPTE format.

In 3rd inlet: Sets the seconds of the current time when **timeout** is in non-bits SMPTE format.

In right inlet: Sets the frame count of the current time when **timeout** is in non-bits SMPTE format.

format — The word format, followed by a number between 0 and 9, sets the **timeout** object's time format. Refer to Arguments below for a list of valid time formats.

bang — Writes the current time to Max's MIDI Manager Clock Out time port.

Arguments

int — Optional. Initializes the time format to be used by the MIDI Manager Clock Out time port. Note that multiple **timeout** objects must use the same format. Exactly why you would need more than one timeout object running simultaneously is not apparent. Time formats are specified by numbers 0-9 as follows:

0	milliseconds
1	beats
2	24 frames/sec bits
3	25 frames/sec bits
4	30 frames/sec Drop bits
5	30 frames/sec bits
6	24 frames/sec
7	25 frames/sec
8	30 frames/sec Drop
9	30 frames/sec.

If no argument is present, the default time format is milliseconds.

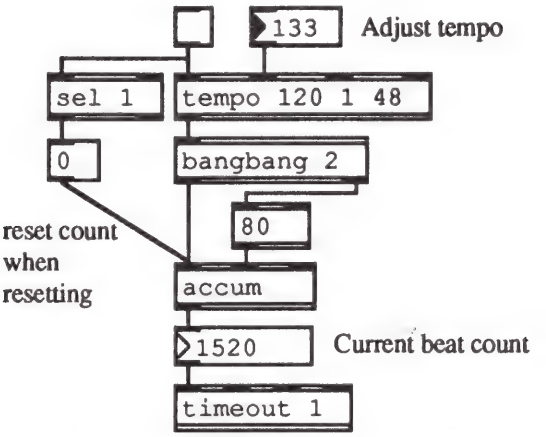
Output

timeout has no outlets. It writes the time received at its inlets to Max's MIDI Manager Clock Out time port. Using the MIDI Manager *PatchBay* application, this port can be connected to other applications, or to the Apple MIDI Driver to send MIDI Clock or MIDI Time Code to synchornize external devices to the Macintosh. **timeout** will not function unless Max is using the MIDI Manager.

Write current time to MIDI Manager time port

timeout

Examples



This patch uses **timeout** *to provide a master tempo in MIDI Manager beats format.*

See Also

MIDI	MIDI overview and specification
Ports	How MIDI ports are specified
timein	Report time from external time code source

timer

Report elapsed time between two events

Input

bang — In left inlet: Starts—or *re*starts—the **timer**.

In right inlet: Sends out the time elapsed since the **timer** was started.

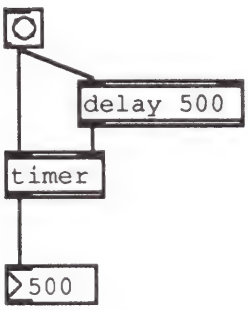
Arguments

None.

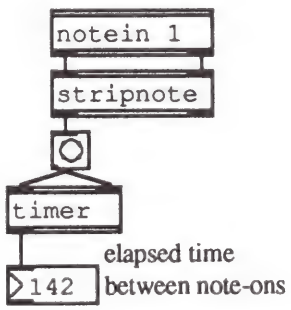
Output

int — When a bang is received in the *right* inlet, the time elapsed, in milliseconds, since the **timer** was started, is sent out the outlet.

Examples



Report time between bangs



One event can report time, then restart **timer**

See Also

clocker — Report elapsed time at regular intervals
Tutorial 20 — Using the Macintosh keyboard

Report zero/ non-zero transitions

TogEdge

Input

int — The number is stored in **TogEdge**. If it is not 0, and the previously stored number was 0, **TogEdge** sends a bang out the left outlet. If the number is 0, and the previously stored number was not 0, **TogEdge** sends a bang out the right outlet. Otherwise, **TogEdge** sends no output.

float — Converted to int.

bang — Switches the value stored in **TogEdge** from 0 to non-zero, or vice versa, and reports the change by sending a bang out one of the outlets.

Arguments

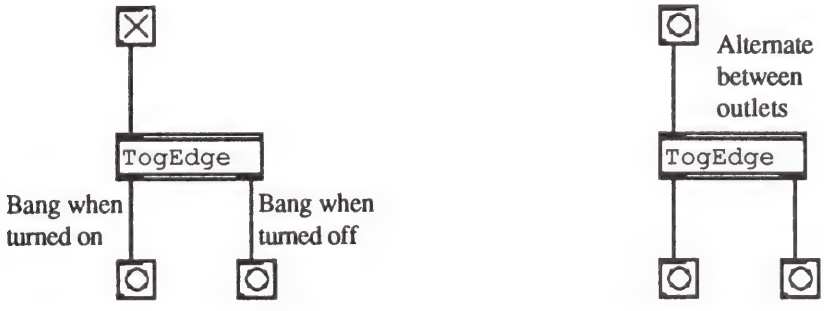
None.

Output

bang — Out left outlet: If the stored value is changed from 0 to not 0.

Out right outlet: If the stored value is changed from not 0 to 0.

Examples



Used as a detector of on/off status, or to switch back and forth between two triggers

See Also

change — Filter out repetitions of a number
led — Display on/off status, in color
toggle — Switch between on and off (1 and 0)

toggle

Switch between on and off (1 and 0)

Input

int — The number is sent out the outlet. If the number is not 0, **toggle** displays an X, showing it is *on*. If it is 0, **toggle** is blank, showing it is *off*.

float — Converted to int.

bang — Switches **toggle** on if it is off; switches it off if it is on.

A mouse click on **toggle** has the same effect as a bang in its inlet.

Arguments

Not applicable.

Output

int — A number received in the inlet is sent out the outlet. A bang or a mouse click sends 1 or 0 out the outlet, depending on whether **toggle** is being turned on or off.

Examples

Used as an onscreen controller, or to display the on/off status of numbers passing through

See Also

led	Display on/off status, in color
TogEdge	Report a change in zero/non-zero values
Tutorial 5	**toggle** and **comment**

Output incoming MIDI aftertouch values

touchin

Input

(MIDI) **touchin** receives its input from MIDI aftertouch (channel pressure) messages received directly from the serial ports, or from the *virtual ports* of the MIDI Manager and/or OMS.

Arguments

a-z Optional. Specifies the port from which to receive incoming aftertouch messages. If there is no argument, **touchin** receives from all channels on all ports.

OMS name Optional. When using OMS, you may use the name of an OMS Input Device as an argument to specify a port instead of a-z.

a-z and int A letter and number combination (separated by a space) indicates a port and a specific MIDI channel on which to receive aftertouch messages. Channel numbers greater than 16 will be *wrapped around* to stay within the 1-16 range.

int A number alone can be used in place of a letter and number combination. Numbers 1-16 specify a channel number on port a, numbers 17-32 specify channels 1-16 on port b, etc.

If the MIDI Manager is being used, a letter a-f specifies a virtual port. If the MIDI Manager is not being used, the letter a specifies the modem port and b specifies the printer port. If OMS is being used, the letter a-z can be an abbreviation for any OMS or MIDI Manager port (configured using the **MIDI Setup...** dialog).

Output

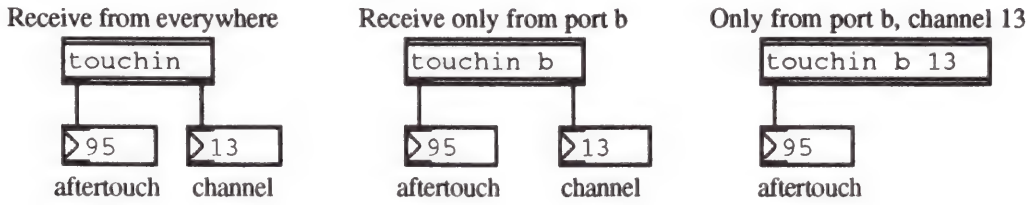
int If a specific channel number is included in the argument, there is only one outlet. The output is the incoming aftertouch value, from 0-127, on the specified channel and port.

If there is no channel number specified by the argument, **touchin** will have a second outlet, on the right, which will output the channel number of the incoming aftertouch message.

touchin

Output incoming MIDI aftertouch values

Examples



Aftertouch messages can be received from everywhere, a specific port, or a specific port and channel

See Also

touchout	Transmit MIDI aftertouch messages
midiin	Output incoming raw MIDI data
OMS	Using Max with OMS
Ports	How MIDI ports are specified
Tutorial 16	More MIDI ins and outs

Transmit MIDI aftertouch messages

touchout

Input

int — In left inlet: The number is transmitted as an aftertouch value on the specified channel and port. Numbers are limited between 0 and 127.

In right inlet: The number is stored as the channel number on which to transmit the aftertouch messages.

float — Converted to int.

list — In left inlet: The first number is the aftertouch value, and the second number is the channel, of a MIDI aftertouch message, transmitted on the specified channel and port.

port — In left inlet: When not using OMS, the word port, followed by a letter a-f, specifies the port used to transmit MIDI aftertouch messages. When using OMS, port can be followed by a-z or the name of an OMS Output Device. In either case, the word port is optional and can be omitted.

Arguments

a-z — Optional. Specifies the port for transmitting MIDI atertouch messages. Channel numbers greater than 16 received in the right inlet will be *wrapped around* to stay within the 1-16 range. If there is no argument, **touchout** initially transmits out port a, on MIDI channel 1.

a-z and int — A letter and number combination (separated by a space) indicates a port and a specific MIDI channel on which to transmit aftertouch messages. Channel numbers greater than 16 will be *wrapped around* to stay within the 1-16 range.

OMS name — Optional. When using OMS, the name of an OMS Output Device may be used as the first argument to specify the port.

int — A number alone can be used in place of a letter and number combination. Numbers 1-16 specify a channel number on port a, numbers 17-32 specify channels 1-16 on port b, etc. In this case, channel numbers received in the right inlet will specify both the port and the channel number.

If the MIDI Manager is being used, a letter a-f specifies a virtual port. If the MIDI Manager is not being used, the letter a specifies the modem port and b specifies the printer port. If OMS is being used, the letter a-z can be an abbreviation for any OMS or MIDI Manager port (configured using the **MIDI Setup...** dialog).

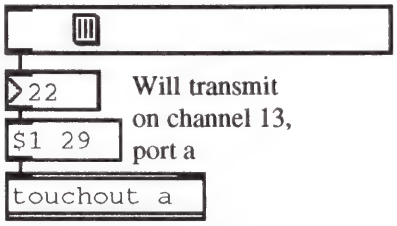
Output

(MIDI) — There are no outlets. The output is a MIDI aftertouch message transmitted directly to the serial ports, or to the *virtual ports* of the MIDI Manager.

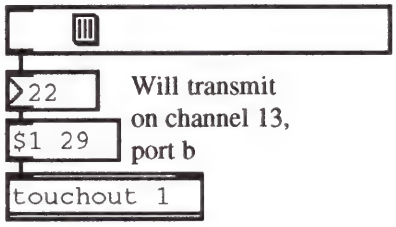
touchout

Transmit MIDI aftertouch messages

Examples



Letter argument transmits to only one port



Otherwise, number specifies both port and channel

See Also

touchin	Ouput incoming MIDI aftertouch values
midiout	Transmit raw MIDI data
Ports	How ports are specified
Tutorial 16	More MIDI ins and outs

Send a number to many places, in order

trigger / t

Input

int or float — The number is sent out each outlet in the form of either an int, a float, or a bang.

bang — Causes either a bang, an integer 0, or a float 0. to be sent out of each outlet.

list — The list is sent out any outlet with the letter l assigned to it. Out other outlets, the list is converted and sent out as integer 0, float 0., or bang.

Arguments

i, f, b, or l — Optional. The number of arguments determines the number of outlets. Each outlet sends out either int, float, bang, or list, depending on the arguments. You can supply up to 10 arguments. If there are no arguments, there are two outlets, both of which send an int.

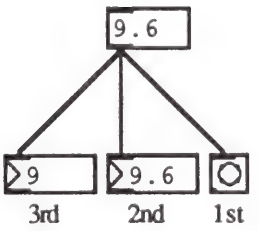
Output

int or float — A number received in the inlet is sent out each outlet, in order from right to left. The number will be converted to int, float, or bang before being sent out, depending on the argument that corresponds to each outlet. A bang received in the inlet will be converted to integer 0 by an i or l outlet, and to float 0. by an f argument. A list received in the inlet will be converted to 0 by an i outlet and 0. by an f outlet.

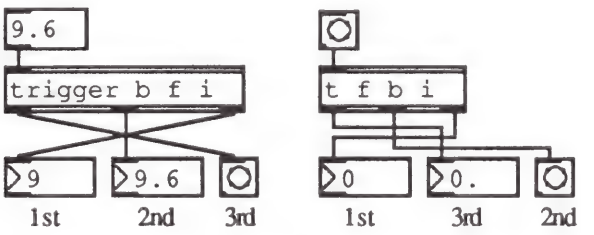
bang — Anything received in the inlet will be converted to bang before being sent out a b outlet.

list — A list received in the inlet will be sent out unchanged by an l outlet.

Examples



Order is normally right-to-left



Any other order can be specified by **trigger**

See Also

bangbang — Send a bang many places, in order
Tutorial 7 — Right-to-left order

Trough

If a number is less than previous numbers, output it

Input

int — In left inlet: If the input is less than the value currently stored in **Trough**, it is stored as the new minimum value and is sent out.

In right inlet: The number is stored in **Trough** as the new minimum value, and is sent out.

float — Converted to int.

list — In left inlet: The second number is stored as the new minimum value and is sent out, then the first number is received in the left inlet.

Arguments

None. The initial value stored in **Trough** is 128.

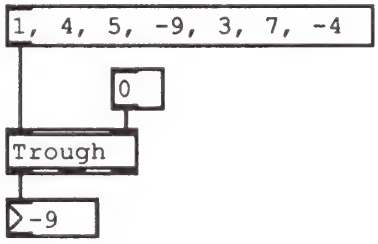
Output

int — Out left outlet: New minimum values are sent out. (Numbers received in the right inlet are *always* the new minimum value.)

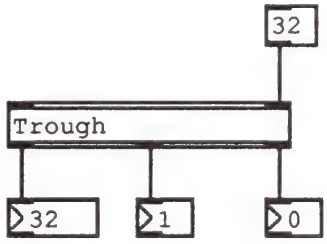
Out middle outlet: If the number received is a new minimum value, the output is 1. If the number received in the left inlet is *not* a new minimum value, the output is 0.

Out right outlet: If the number received is a new minimum value, the output is 0. If the number received in the left inlet is *not* a new minimum value, the output is 1.

Examples



Find the smallest in a series of numbers



Number in right inlet always sets a new trough

See Also

minimum	Output the lesser of two numbers
Peak	If a number is greater than previous numbers, output it
<	*Is less than*, comparison of two numbers

Transparent button, sends a bang

ubutton

Input

bang — When a bang is received in its inlet, **ubutton** flashes briefly and sends a bang out its left outlet. A mouse click on **ubutton** inverts the button for as long as the mouse is held down, and sends a bang out the right outlet when the mouse button is pressed down, and another bang out the left outlet when the mouse button is released.

any symbol — Converted to bang.

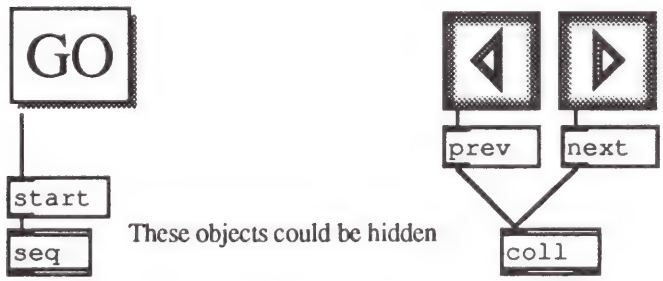
stay — The word stay, followed by a number, sets **ubutton** to wait for that particular number. When that number is received in the inlet, no output is sent, but **ubutton** remains highlighted until some *other* message (or a mouse click) is received. A message of stay 0 takes the **ubutton** out of "stay" mode—it no longer looks for any particular number.

int — If **ubutton** is in "stay" mode and the number matches the stay value the **ubutton** is looking for, the button is highlighted. Otherwise the button is unhighlighted (or remains that way). If **ubutton** is not in stay mode, a number is the same as bang.

Output

bang — A bang in the inlet causes **ubutton** to flash and send a bang out the left outlet. A mouse click sends a bang out the right outlet, and a bang is sent out the left outlet when the mouse is released.

Examples



When **ubutton** *is placed on comments or pictures, they seem to "respond" to a mouse click*

See Also

bangbang — Send a bang to many places, in order
button — Flash on any message, send a bang
Tutorial 19 — Screen aesthetics

unpack

Break a list up into individual numbers

Input

list — Each number in the list (up to the number of outlets) is sent out the outlet corresponding to its position in the list. **unpack** does not store the list it receives.

int — The number is sent out the left outlet.

float — Converted to int, unless the outlet was initialized with a float argument. The number is sent out the left outlet.

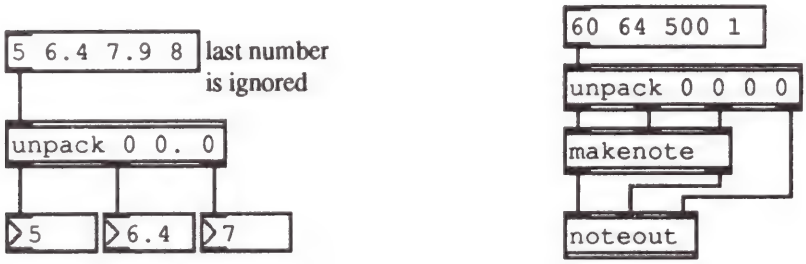
Arguments

int or float — Optional. The number of outlets is determined by the number of arguments, up to 10. The arguments can either be numbers, whose types determine the type of outlet (int or float), or i for an int outlet and f for a float outlet.

Output

int — Each outlet sends out a number from the list received in the inlet. The first number in the list is sent out the leftmost outlet, and so on.

Examples



Each item in a list can be sent to a different place

See Also

iter	Break a list up into a series of numbers
pack	Combine numbers into a list
Tutorial 30	Number groups

Output numbers by moving a slider onscreen

uslider

Input

int — The number received in the inlet is displayed graphically by **uslider**, and is passed out the outlet. Optionally, **uslider** can multiply the number by some amount and add an offset to it, before sending it out the outlet.

The **uslider** will also send out numbers in response to mouse clicking or dragging.

float — Converted to int.

bang — Sends out the number currently stored in **uslider**.

set — The word set, followed by a number, resets the value displayed by **uslider**, without triggering output.

size — The word size, followed by a number, resets the range of the **uslider**. A size of 1 disables the **uslider**, graying it out. Any number less than 1 will be set to 2.

Arguments

The range of **uslider** is set by selecting it (when the Patcher window is unlocked) and choosing **Get Info...** from the Max menu. Numbers received in the inlet are automatically limited between 0 and the number 1 less than the specified range.

The **Get Info...** dialog box also provides a *Multiplier*—by which all numbers will be multiplied before being sent out, and an *Offset*—which will be added to the number, after multiplication. A newly created **uslider** has a range of 128, a multiplier of 1, and an offset of 0.

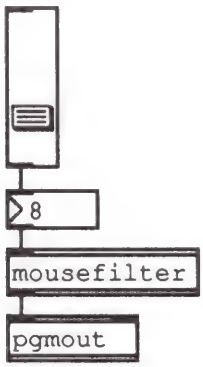
Output

int — Numbers received in the inlet, or produced by clicking or dragging on **uslider** with the mouse, are first multiplied by the multiplier, then have the offset added to them, then are sent out the outlet.

uslider

Output numbers by moving a slider onscreen

Examples



Produce output by dragging onscreen...

or use to display numbers passing through

See Also

hslider, slider	Output numbers by moving a slider onscreen
Tutorial 14	Sliders and dials

Send a specific number of bangs

Uzi

Input

bang — In left inlet: Begins sending out bang messages, one after another. The number of bangs to send is determined by the last number received in the left or right inlets.

int — In left inlet: Sets the number of bangs to send, then begins sending them out, one after another.

In right inlet: Sets the number of bangs to send without producing output.

Arguments

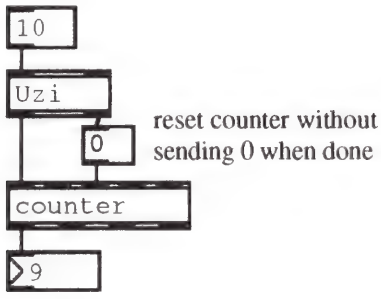
None.

Output

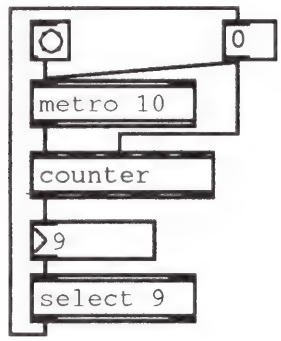
bang — Out left outlet: When **Uzi** receives a bang or int in its left inlet, a set number of bangs are sent out, one after another.

bang — Out right outlet: After the last bang is sent out its left outlet, **Uzi** sends one bang out its right outlet. This can be used a signal that the specified number of bangs have been sent, much like the "carry" outlet on the **counter** object.

Examples



Counting quickly from 0 to 9 using **Uzi**



The same thing using **metro**

See Also

metro — Output bangs at regular intervals
counter — Count the bangs received, output the count

This page intentionally left blank

Share a stored number with other objects

value / v

Input

int or float — The number is stored, to be shared by all other **value** objects with the same name, even if they are in another patch. A number received in any other **value** object with the same argument will change the stored value.

bang — Sends out the stored number.

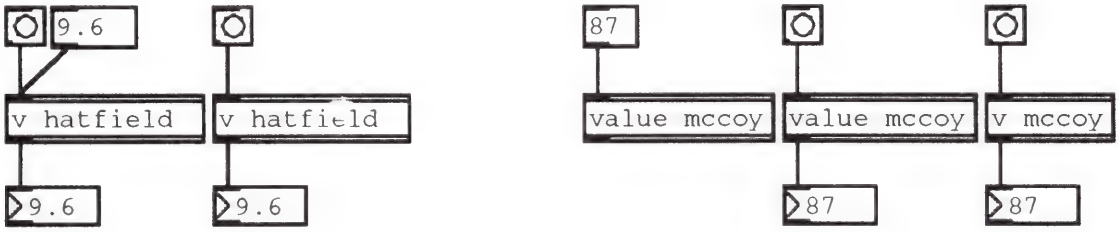
Arguments

any symbol — Obligatory. Gives a name to **value**.

Output

int or float — A bang in the inlet causes the stored value to be sent out.

Examples



One value is shared between all **value** *objects which share the same name*

See Also

send — Send messages, without patch cords
receive — Receive messages, without patch cords
Tutorial 24 — **send** and **receive**

vdp

Control videodisc player through serial port

Input

The videodisc control object attempts to implement most of the "lower level" commands in the HyperCard Interactive Video Toolkit. Each command sends a stream of numbers out the object's left outlet, intended to be connected to the **serial** object. The description of each command below discusses what effect the command has on the player, not the exact character stream sent by **vdp**. Because videodisc players have relatively buffer-less serial interfaces, **vdp** places each command it receives in a queue, and sends it out when the player has finished executing its most recent command. This "feature" may cause a delay between the time a command is sent to the **vdp** object and the time it is actually sent out the serial port.

clear — Removes any pending commands from the queue and resets the object.

control — The word control, followed by a number, tells the player to perform one of the following operations:

Number	*Operation*
0	Initialize and reset player
1	Eject disk
2	Audio off
3	Audio 1 on
4	Audio 2 on
5	Stereo on
6	Picture on
7	Picture off
8	Display frame numbers on
9	Display frame numbers off
11	Frame access mode
12	Time access mode
13	Chapter access mode

fps — Sets the playing speed. The fps message is followed by a number (frames per second) or an adjective. The following adjectives and numbers are equivalent (at least for the Pioneer 4200):

slowest	1
slower	10
slow	15
normal	30
fast	60
faster	90
fastest	120

frame — Asks the player what its current frame number is and sends the response (received in the right inlet) out the middle-right outlet.

int — In left inlet: Same as play from a frame number to the end of the disc.

In right inlet: **vdp** expects responses from the player to be fed from the **serial** object into its right inlet. When **vdp** sees "received" (letter R followed by the return character) from the player, it sends the next command from its queue of pending commands. The example shows how to connect the **vdp** and **serial** objects together.

play — With no arguments, play starts playing at the current speed from the current location to the end of the disk (or until the player receives another command). With one argument (a frame number), play searches to the specified frame number and begins playing to the end of the disk. With two arguments, play searches to the location specified by the first number and plays until the disc reaches the second frame number.

scan — Initiates a "fast forward" or "rewind" operation. scan forward moves forward, scan backward moves backward.

search — The first argument indicates a frame number to search to. The second, optional argument, if non-zero, instructs the player to keep the picture on while searching. If searching a great distance from the current location, the player may not be able to keep from blanking the screen. Once the player arrives at the desired frame, it will display the (still) image from that frame.

step — Followed by -1, step pauses the player (if playing) and displays the *previous* frame. Followed by 1, step pauses the player (if playing) and displays the *next* frame.

stop — Pauses the player.

Arguments

Optional. Videodisc player model. See **vdp** help file for models currently supported. Default is Pioneer 4200.

Output

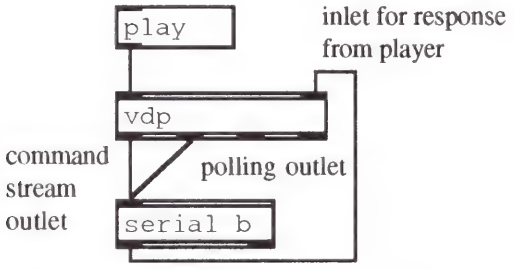
int — Out left outlet: A stream of characters, coded instructions to the videodisc player, for each command. These numbers are intended to be sent to the left inlet of a **serial** object.

bang — Out middle-left outlet: After sending a command out its left outlet, **vdp** begins "polling" the **serial** object for a response from the player by sending bang messages out this outlet approximately every 20 milliseconds, until **vdp** receives a "received" signal from the player in its right inlet. A bang sent to a **serial** object causes any characters recevied to be sent out the **serial** object's outlet.

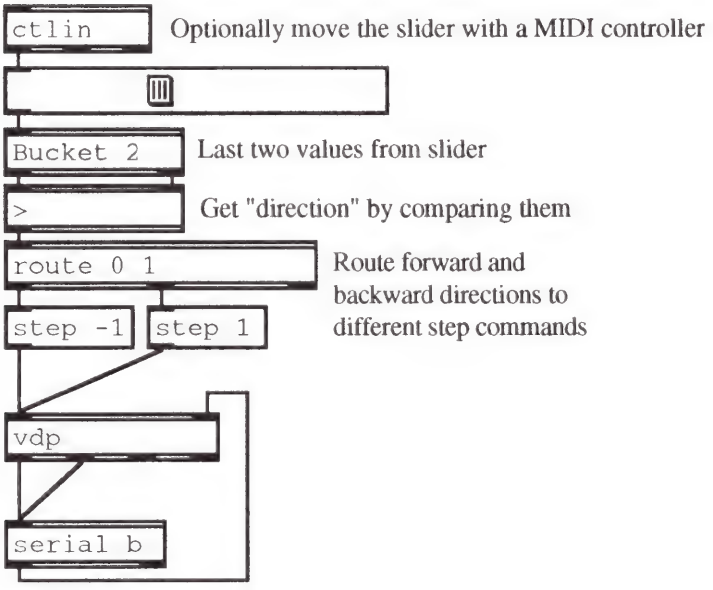
int — Out middle-right outlet: Current frame number, received from the player in response to a frame message.

int Out right outlet: Not implemented.

Examples



Basic configuration of **vdp** *and* **serial** *objects*



Using **Bucket** *to implement "scrubbing" with a slider or MIDI controller*

See Also

serial Send and receive characters from serial ports and cards
Pioneer 4200 operation manual
HyperCard Interactive Video Toolkit documentation

Share a stored number with other objects

value / v

Input

int or float — The number is stored, to be shared by all other **value** objects with the same name, even if they are in another patch. A number received in any other **value** object with the same argument will change the stored value.

bang — Sends out the stored number.

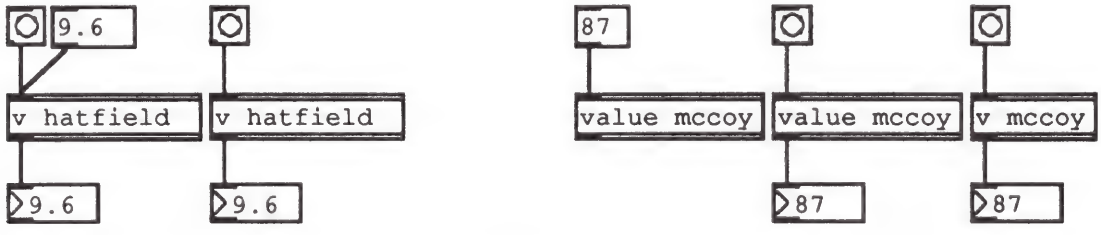
Arguments

any symbol — Obligatory. Gives a name to **value**.

Output

int or float — A bang in the inlet causes the stored value to be sent out.

Examples



One value is shared between all **value** *objects which share the same name*

See Also

send — Send messages, without patch cords
receive — Receive messages, without patch cords
Tutorial 24 — **send** and **receive**

xbendin

Interpret extra precision MIDI pitch bend values

Input

int — The numbers are individual bytes of a MIDI message stream, received from an object such as **midiin** or **seq**. MIDI pitch bend messages are recognized by **xbendin**, and the pitch bend data is sent out in full precision.

Arguments

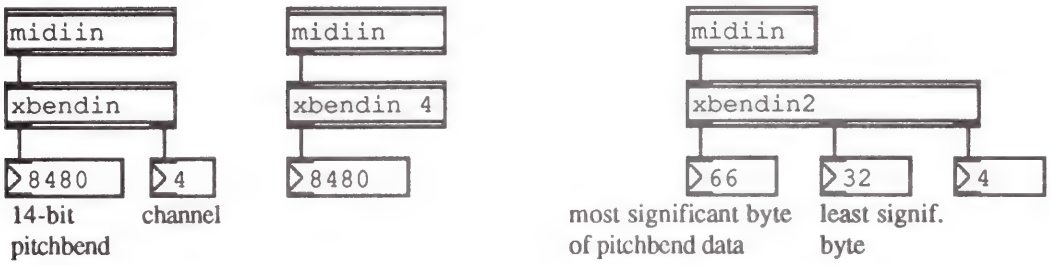
int — Optional. The number specifies a MIDI channel on which to recognize pitch bend messages. If there is no argument, **xbendin** recognizes pitch bend messages on all channels, and the channel number is sent out the extra outlet on the right.

xbendin2 — Optional. Normally, **xbendin** sends pitch bend values out the left outlet as 14-bit values. If the object is called **xbendin2**, however, there will be an additional outlet. The most significant data byte of the message is sent out the leftmost outlet, and the least significant data byte is sent out the second outlet.

Output

int — The pitch bend value is sent out the left outlet of **xbendin** as a single 14-bit value. If the object is called **xbendin2**, there is an additional outlet. The most significant 7 bits are sent out the leftmost outlet, and the least significant (extra precision) 7 bits are sent out the second outlet. If there is no channel number specified as an argument (omni on), **xbendin** will have an extra outlet on the right, which will output the channel number of the incoming pitch bend message.

Examples



Pitch bend values are sent out as a single number or as two separate bytes

See Also

bendin	Output incoming MIDI pitch bend values
midiin	Output incoming raw MIDI data
xbendout	Format extra precision MIDI pitch bend messages
Tutorial 34	Managing raw MIDI data

Format extra precision
MIDI pitch bend messages

xbendout

Input

int — In left inlet: The number is a 14-bit pitch bend value to be formatted into a complete MIDI pitch bend message by **xbendout**.

In right inlet: The number is stored as the MIDI channel for the pitch bend message sent out by **xbendout**. Channel numbers greater than 16 will be *wrapped around* to stay within the 1-16 range.

list — The first number is a 14-bit pitch bend value, and the second number is the channel. Both numbers are stored and are formatted into a MIDI pitch bend message which is sent out the outlet.

bang — Sends out a MIDI pitch bend message using the numbers currently stored in **xbendout**.

Arguments

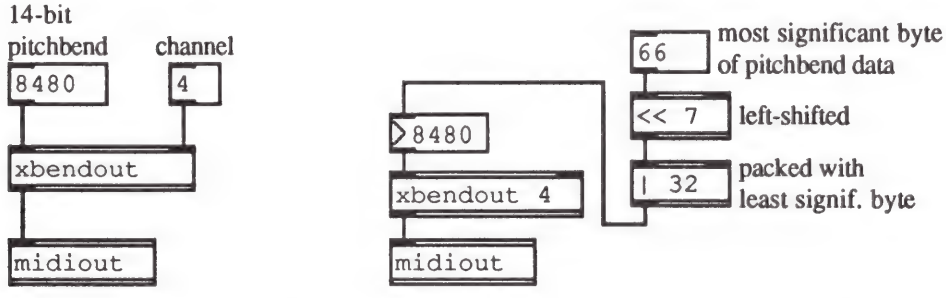
xbendout2 — If the object is called **xbendout2**, there will be three inlets. The most signficant byte of the pitch bend message is received in the left inlet, and the least significant (extra precision) byte is received in the middle inlet.

int — Optional. The number sets an initial value for the MIDI channel of the pitch bend messages. If there is no argument, the initial channel number is 1.

Output

int — When a pitch bend value is received in the left inlet, the complete MIDI pitch bend message is sent out the outlet, byte-by-byte.

Examples



14-bit pitch bend value is formatted into a MIDI message, which is sent out byte-by-byte

xbendout

Format extra precision MIDI pitch bend messages

See Also

bendout	Transmit MIDI pitch bend messages
midiout	Transmit raw MIDI data
xbendin	Interpret extra precision MIDI pitch bend values
Tutorial 34	Managing raw MIDI data

Interpret MIDI note messages with release velocity

xnotein

Input

int The numbers are individual bytes of a MIDI stream from **midiin**. Whereas a note-on with a velocity of 0 is most commonly used to indicate a note-off, **xnotein** also recognizes the MIDI note-off command, and outputs its release velocity.

Arguments

int Optional. Specifies a channel number to look for incoming MIDI note-on and note-off messages. Channel numbers greater than 16 will be *wrapped around* to stay within the 1-16 range. If there is no argument, **xnotein** recognizes note-on and note-off messages on all MIDI channels, and the channel number of the message is sent out the rightmost outlet.

Output

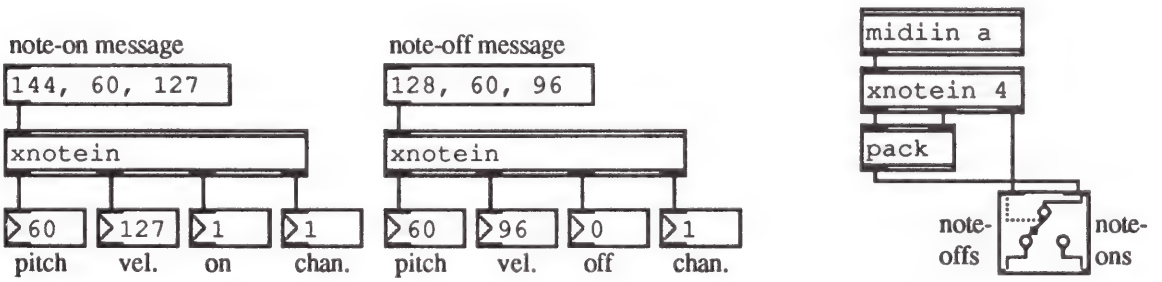
int Out left outlet: The pitch value of the incoming note-on or note-off message.

Out 2nd outlet: The key-down or key-up velocity of a note-on or a note-off message.

Out 3rd outlet: The number is the indicator of whether the incoming MIDI message is a note-on or a note-off. If the incoming message is a note-on, the output is 1. If the incoming message is a note-off—or a note-on with a velocity of 0—the output is 0.

If no channel number is specified as an argument, **xnotein** has a 4th outlet on the right. The channel number of incoming messages is sent out the rightmost outlet.

Examples



Both note-on and note-off messages are interpreted, with a key-down or key-up velocity

See Also

notein Output incoming MIDI note values

xnotein

Interpret MIDI note messages with release velocity

midiin	Receive raw MIDI data
xnoteout	Format MIDI note messages with release velocity
Tutorial 34	Managing raw MIDI data

Format MIDI note messages with release velocity

xnoteout

Input

int — In left inlet: The number is used as the pitch value for a note-on or note-off message, and the message is sent out the oultet byte-by-byte.

In left-middle inlet: The number is stored as the velocity for either a note-on or a note-off message. If no number has been received yet, the velocity for note-ons is 64, and the velocity for note-offs is 0.

In right-middle inlet: The number is stored as the indicator of whether outgoing messages should be note-ons or note-offs. If the number is not 0, **xnoteout** will send out a note-on message. If the number is 0, **xnoteout** will send out a note-off message with a release velocity. If no number has been received yet, it is initially 1 (note-on).

In right inlet: The number is stored as the channel for the MIDI message sent out by **xnoteout**. Channel numbers greater than 16 will be *wrapped around* to stay within the 1-16 range.

float — Converted to int.

list — The first number is the pitch value, the second number is the velocity, the third number is the note-on/note-off indicator (non-zero for note-on, 0 for note-off), and the fourth number is the channel. The numbers are stored by **xnoteout**, and a MIDI note-on or note-off message is sent out.

bang — Sends out a MIDI message using the numbers currently stored in **xnoteout**.

Arguments

int — Optional. Sets an initial value for the MIDI channel of the outgoing messages. Channel numbers greater than 16 will be *wrapped around* to stay within the 1-16 range. If there is no argument, the initial channel number is 1.

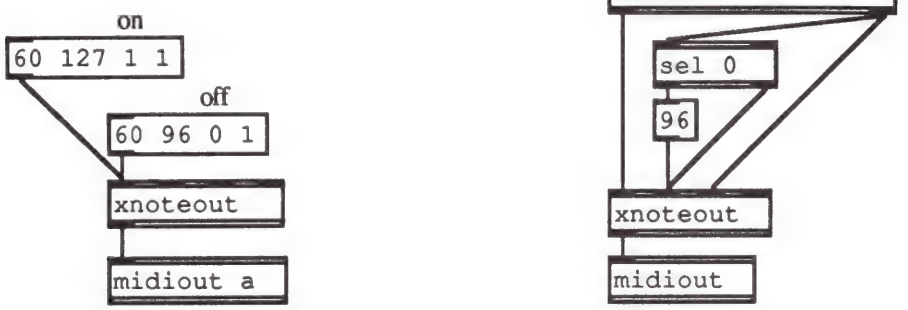
Output

int — When a pitch value is recieved, a complete MIDI note-on or note-off message is sent out the outlet , byte-by-byte. Whereas a note-on with a velocity of 0 is most commonly used to indicate a note-off, **xnoteout** sends out the MIDI note-off command and uses the specified velocity as a *release* velocity.

xnoteout

Format MIDI note messages with release velocity

Examples



The numbers are formatted into a MIDI note-on or note-off message, which is sent out byte-by-byte

See Also

noteout	Transmit MIDI note messages
midiout	Transmit raw MIDI data
xnotein	Interpret MIDI note messages with release velocity
Tutorial 34	Managing raw MIDI data

Add two numbers, output the result

+

Input

int — In left inlet: The number is added to the number in the right inlet, and the result is sent out the outlet.

In right inlet: The number is stored for addition to a number received in the left inlet.

float — Converted to int, unless + has a float argument.

bang — In left inlet: Performs the addition with the numbers currently stored. If there is no argument, + initially holds 0.

list — In left inlet: The first number is added to the second number, and the result is sent out the outlet.

Arguments

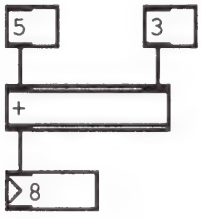
int or float — Optional. Sets the initial value, to be added to a number received in the left inlet. Argument with a decimal point causes the numbers to be added as floats.

Output

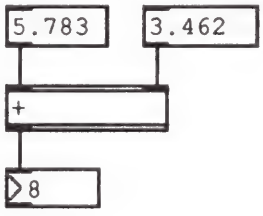
int — The sum of the two numbers received in the inlets.

float — Only if there is an argument with a decimal point.

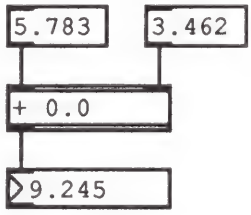
Examples



Normally adds ints



Floats are truncated before addition...



unless there is a float argument

See Also

expr — Evaluate a mathematical expression
Tutorial 8 — Doing math in Max

Input

int	In left inlet: The number in the right inlet is subtracted from the number, and the result is sent out the outlet. In right inlet: The number is stored, to be subtracted from a number received in the left inlet.
float	Converted to int, unless - has a float argument.
bang	In left inlet: Performs the subtraction with the numbers currently stored. If there is no argument, - initially holds 0.
list	In left inlet: The second number is subtracted from the first number, and the result is sent out the outlet.

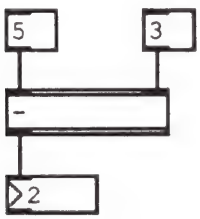
Arguments

int or float	Optional. Sets the initial value, to be subtracted from a number received in the left inlet. Argument with a decimal point causes the numbers to be subtracted as floats.

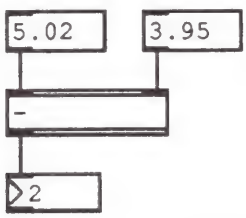
Output

int	The difference between the two numbers received in the inlets.
float	Only if there is an argument with a decimal point.

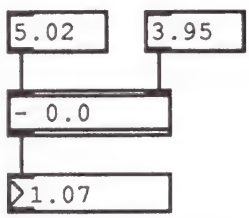
Examples



Subracted as ints



Floats are truncated before subtractiou...



...unless there is a float argument

See Also

expr	Evaluate a mathematical expression
Tutorial 8	Doing math in Max

Multiply two numbers, output the result

*

Input

int — In left inlet: The number is multiplied by the number in the right inlet, and the result is sent out the outlet.

In right inlet: The number is stored for multiplication with a number received in the left inlet.

float — Converted to int before multiplication, unless * has a float argument.

bang — In left inlet: Performs the multiplication with the numbers currently stored. If there is no argument, * initially holds 0 as a multiplier.

list — In left inlet: The first number is multiplied by the second number, and the result is sent out the outlet.

Arguments

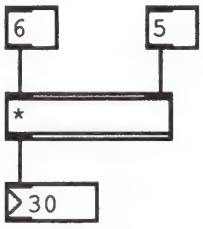
int or float — Optional. Sets the initial value, to be multiplied by a number received in the left inlet. Argument with a decimal point causes the numbers to be multiplied as floats.

Output

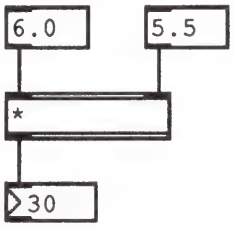
int — The product of the two numbers received in the inlets.

float — Only if there is an argument with a decimal point.

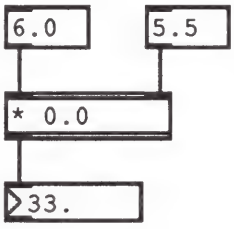
Examples



Multiplied as ints



Floats are truncated before multiplication...



...unless there is a float argument

See Also

expr	Evaluate a mathematical expression
<<	Shift all bits to the left
Tutorial 8	Doing math in Max

/

Divide two numbers, output the result

Input

int In left inlet: The number is divided by the number in the right inlet, and the result is sent out the outlet.

In right inlet: The number is stored as the *divisor* (the number to be divided into the number in the left inlet).

float Converted to int, unless / has a float argument.

bang In left inlet: Performs the division with the numbers currently stored.

list In left inlet: The first number is divided by the second number, and the result is sent out the outlet.

Arguments

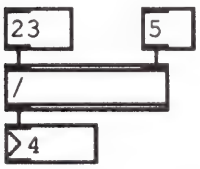
int or float Optional. Sets an initial value for the divisor. If there is no argument, the divisor is set to 1 intially. Argument with a decimal point causes the numbers to be divided as floats. (Division by 0 is not allowed. Int division by 0 will have the same result as dividing by 1. Float division by 0 will always cause an output of -2^{31}.)

Output

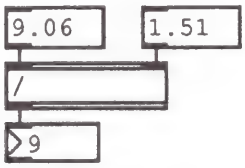
int When the two numbers in the inlets are divided, the result is sent out the outlet.

float Only if there is an argument with a decimal point.

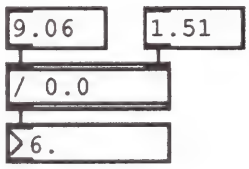
Examples



Remaineder is discarded



Floats are truncated before division...



...unless there is a float argument

See Also

expr	Evaluate a mathematical expression
%	Divide two numbers, output the remainder
Tutorial 8	Doing math in Max

Divide two numbers, output remainder

%

Input

int — In left inlet: The number is divided by the number in the right inlet, and the *remainder* is sent out the outlet.

In right inlet: The number is stored as the *divisor* (the number to be divided into the number in the left inlet) for calculating the remainder.

float — Converted to int.

bang — In left inlet: Performs the operation with the numbers currently stored.

list — In left inlet: The first number is divided by the second number, and the remainder is sent out the outlet.

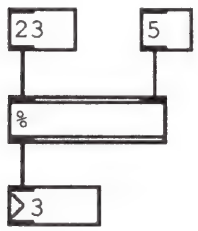
Arguments

int — Optional. Sets an initial value for the divisor. If there is no argument, the divisor is set to 1 initially.

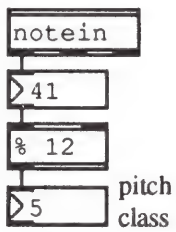
Output

int — When the two numbers in the inlets are divided, the remainder is sent out the outlet. **%** is sometimes called the *modulo operator*.

Examples



Output is the remainder



Output is from 0 to 11

See Also

expr	Evaluate a mathematical expression
/	Divide two numbers, output the result
Tutorial 8	Doing math in Max

<

Is less than, *comparison of two numbers*

Input

int — In left inlet: If the number *is less than* the number in the right inlet, < outputs 1. Otherwise, < outputs 0.

In right inlet: The number is stored to be compared with a number received in the left inlet.

float — Converted to int before comparison, unless < has a float argument.

bang — In left inlet: Performs the comparison with the numbers currently stored. If there is no argument, < initially holds 0 for comparison.

list — In left inlet: If the first number *is less than* the second number, < outputs 1. Otherwise, < outputs 0.

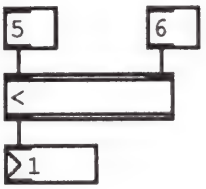
Arguments

int or float — Optional. Sets the initial value, to be compared with a number received in the left inlet. Argument with a decimal point forces a float comparison.

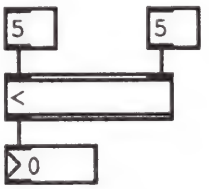
Output

int — 1 if the number in the left inlet is less than the number in the right inlet. 0 if the number in the left inlet *is greater than or equal to* the number in the right inlet.

Examples



Number on left is less than number on right



Number on left is not less than number on right

See Also

<= — *Is less than or equal to,* comparison of two numbers
>= — *Is greater than or equal to,* comparison of two numbers
Tutorial 15 — Making decisions with comparisons

Is less than or equal to, *comparison of two numbers*

<=

Input

int — In left inlet: If the number *is less than or equal to* the number in the right inlet, <= outputs 1. Otherwise, <= outputs 0.

In right inlet: The number is stored to be compared with a number received in the left inlet.

float — Converted to int before comparison, unless <= has a float argument.

bang — In left inlet: Performs the comparison with the numbers currently stored. If there is no argument, <= initially holds 0 for comparison.

list — In left inlet: If the first number *is less than or equal to* the second number, <= outputs 1. Otherwise, <= outputs 0.

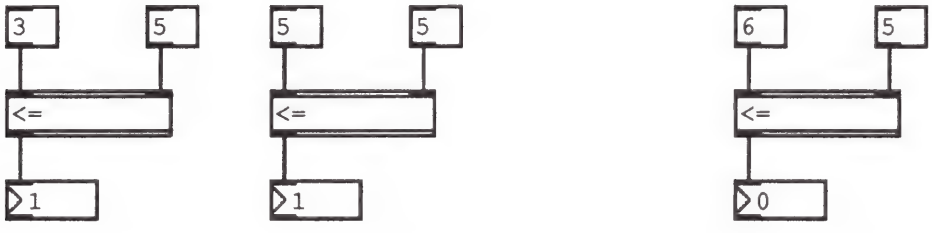
Arguments

int or float — Optional. Sets the initial value, to be compared with a number received in the left inlet. Argument with a decimal point forces a float comparison.

Output

int — 1 if the number in the left inlet is less than or equal to the number in the right inlet. 0 if the number in the left inlet *is greater than* the number in the right inlet.

Examples



Is less than... *or equal to* *Is not less than or equal to*

See Also

<	*Is less than,* comparison of two numbers
>	*Is greater than,* comparison of two numbers
Tutorial 15	Making decisions with comparisons

==

Compare two numbers, output 1 if they are equal

Input

int — In left inlet: The number is compared with the number in the right inlet. If the two numbers are equal, == outputs 1. If they are not equal == outputs 0.

In right inlet: The number is stored to be compared with a number received in the left inlet.

float — Converted to int before comparison, unless == has a float argument.

bang — In left inlet: Performs the comparison with the numbers currently stored. If there is no argument, == initially holds 0 for comparison.

list — In left inlet: Compares first and second number, outputs 1 if they are equal, 0 if they are not equal.

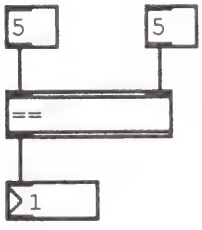
Arguments

int or float — Optional. Sets the initial value, to be compared with a number received in the left inlet. Argument with a decimal point forces a float comparison.

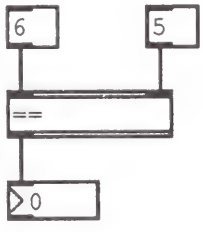
Output

int — 1 if the numbers in the inlets are equal, 0 if they are not equal.

Examples



The numbers are equal



The numbers are not equal

See Also

select — Select certain inputs, pass the rest on
split — Look for a range of numbers
!= — Compare two numbers, output 1 if they are not equal
Tutorial 15 — Making decisions with comparisons

Compare two numbers, output 1 if they are not equal

Input

int — In left inlet: The number is compared with the number in the right inlet. If the two numbers are not equal, != outputs 1. If they are equal != outputs 0.

In right inlet: The number is stored, to be compared with a number received in the left inlet.

float — Converted to int before comparison, unless != has a float argument.

bang — In left inlet: Performs the comparison with the numbers currently stored. If there is no argument, != initially holds 0 for comparison.

list — In left inlet: Compares first and second number, outputs 1 if they are not equal, 0 if they are equal.

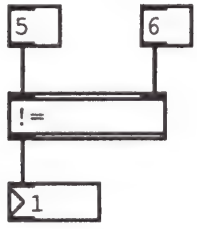
Arguments

int or float — Optional. Sets the initial value, to be compared with a number received in the left inlet. Argument with a decimal point forces a float comparison.

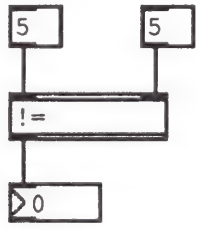
Output

int — 1 if the numbers in the inlets are not equal, 0 if they are equal.

Examples



The numbers are not equal



The numbers are equal

See Also

select	Select certain inputs, pass the rest on
split	Look for a range of numbers
==	Compare two numbers, output 1 if they are equal
Tutorial 15	Making decisions with comparisons

>=

Is greater than or equal to,
comparison of two numbers

Input

int — In left inlet: If the number *is greater than or equal to* the number in the right inlet, >= outputs 1. Otherwise, >= outputs 0.

In right inlet: The number is stored to be compared with a number received in the left inlet.

float — Converted to int before comparison, unless >= has a float argument.

bang — In left inlet: Performs the comparison with the numbers currently stored. If there is no argument, >= initially holds 0 for comparison.

list — In left inlet: If the first number *is greater than or equal to* the second number, >= outputs 1. Otherwise, >= outputs 0.

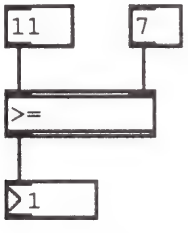
Arguments

int or float — Optional. Sets the initial value, to be compared with a number received in the left inlet. Argument with a decimal point forces a float comparison.

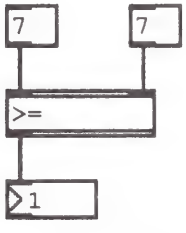
Output

int — 1 if the number in the left inlet is greater than or equal to the number in the right inlet. 0 if the number in the left inlet *is less than* the number in the right inlet.

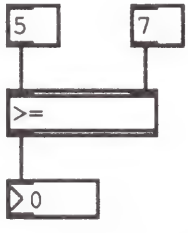
Examples



Is greater than...



or equal to



Is not greater than or equal to

See Also

<	*Is less than,* comparison of two numbers
>	*Is greater than,* comparison of two numbers
Tutorial 15	Making decisions with comparisons

Is greater than, *comparison of two numbers* >

Input

int — In left inlet: If the number *is greater than* the number in the right inlet, > outputs 1. Otherwise, > outputs 0.

In right inlet: The number is stored to be compared with a number received in the left inlet.

float — Converted to int before comparison, unless > has a float argument.

bang — In left inlet: Performs the comparison with the numbers currently stored. If there is no argument, > initially holds 0 for comparison.

list — In left inlet: If the first number *is greater than* the second number, > outputs 1. Otherwise, > outputs 0.

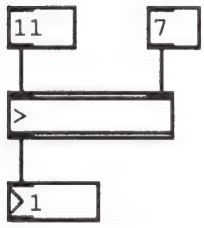
Arguments

int or float — Optional. Sets the initial value, to be compared with a number received in the left inlet. Argument with a decimal point forces a float comparison.

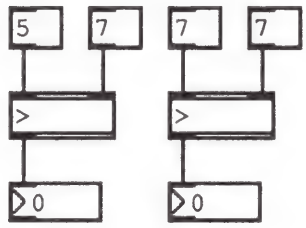
Output

int — 1 if the number in the left inlet is greater than the number in the right inlet. 0 if the number in the left inlet *is less than or equal to* the number in the right inlet.

Examples



The number on the left is greater



The number on the left is not greater

See Also

<= — *Is less than or equal to,* comparison of two numbers
>= — *Is greater than or equal to,* comparison of two numbers
Tutorial 15 — Making decisions with comparisons

&&

If both numbers are non-zero, output 1, otherwise output 0

Input

int — If the number in *both* inlets is not 0, then the output is 1. If the number in one or both of the inlets is 0, then the output is 0. A number in the left inlet triggers the output.

float — Converted to int.

bang — In left inlet: Performs the operation with the numbers currently stored. If there is no argument, **&&** initially holds 0.

list — In left inlet: If both the first and second numbers are not 0, then the output is 1. Otherwise, the output is 0.

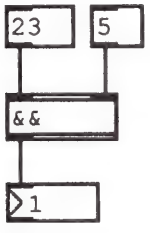
Arguments

int — Optional. Sets an initial value to be stored by **&&**. A number in the right inlet changes the value set by the argument.

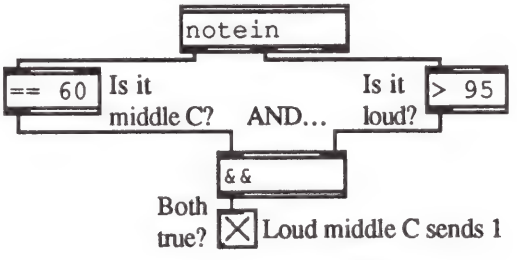
Output

int — If the number in the left inlet *and* the number in the right inlet (or specified by the argument) are both not 0, then the output is 1. Otherwise, the output is 0.

Examples



Both numbers are not 0



Used to combine comparisons

See Also

||	If either number is not 0, output is 1
&	Bitwise "and" of two numbers
Tutorial 15	Making decisions with comparisons

If either of two numbers is non-zero, output 1

Input

int — If the number in *either* inlet is not 0, then the output is 1. If the number in *both* of the inlets is 0, then the output is 0. A number in the left inlet triggers the output.

float — Converted to int.

bang — In left inlet: Performs the operation with the numbers currently stored. If there is no argument, || initially holds 0.

list — In left inlet: If either the first or second number is not 0, then the output is 1. Otherwise, the output is 0.

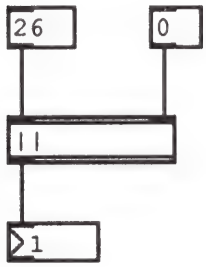
Arguments

int — Optional. Sets an initial value to be stored by ||. A number in the right inlet changes the value set by the argument.

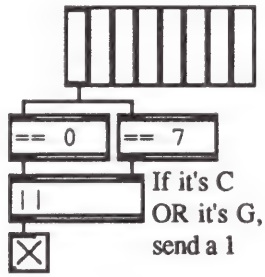
Output

int — If either the number in the left inlet *or* the number in the right inlet (or specified by the argument) is not 0, then the output is 1. Otherwise, the output is 0.

Examples



One of the numbers is not 0



Used to combine comparisons

See Also

&&	If both numbers are not 0, output 1
|	Bitwise "or" of two numbers
Tutorial 15	Making decisions with comparisons

&

Bitwise intersection of two numbers

Input

int — In left inlet: The number is compared, in binary form, with the number in the right inlet. The output is a number comprised of those bits which are 1 in *both* numbers.

In right inlet: The number is stored for comparison with a number received in the left inlet.

float — Converted to int.

bang — In left inlet: Performs the comparison with the numbers currently stored. If there is no argument, **&** initially holds 0 for comparison.

list — In left inlet: Compares the first and second numbers bit-by-bit, and outputs a number comprised of those bits which are 1 in both numbers.

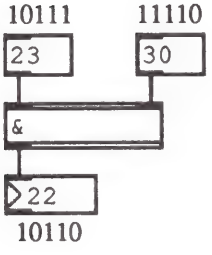
Arguments

int — Optional. Sets an initial value to be compared with a number received in the left inlet.

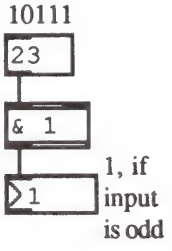
Output

int — The two numbers received in the inlets are compared, one bit at a time. If a bit is 1 in both numbers, it will be 1 in the output number, otherwise it will be 0 in the output number.

Examples



Non-zero bits shared by both numbers



Can be used as an odd/even detector

See Also

&& — If both numbers are not 0, output 1
| — Bitwise union of two numbers

Bitwise "or" of two numbers

Input

int — In left inlet: Outputs a number comprised of all those bits which are 1 in either of the two numbers.

In right inlet: The number is stored for combination with a number received in the left inlet.

float — Converted to int.

bang — In left inlet: Performs the calculation with the numbers currently stored. If there is no argument, I initially holds 0.

list — In left inlet: Combines the first and second numbers bit-by-bit, and outputs a number comprised of all those bits which are 1 in either of the two numbers.

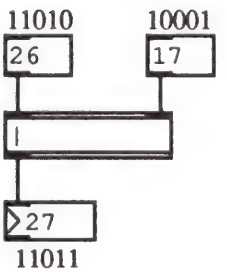
Arguments

int — Optional. Sets an initial value to be or-edwith a number received in the left inlet.

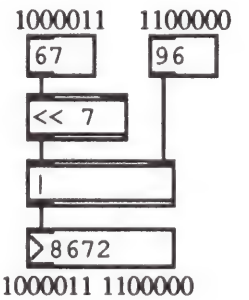
Output

int — All the non-zero bits of the two numbers received in the inlets are combined. If a bit is 1 in either one of the numbers, it will be 1 in the output number, otherwise it will be 0 in the output number.

Examples



All non-zero bits are combined



Can be used to pack two numbers into one int

See Also

I	If either of two numbers is non-zero, output 1
&	Bitwise intersection of two numbers

<<

Shift all bits to the left

Input

int — In left inlet: All bits of the number, in binary form, are shifted to the left by a certain number of bits. The resulting number is sent out the outlet.

In right inlet: The number is stored as the number of bits to left-shift the number in the left inlet.

float — Converted to int.

bang — In left inlet: Performs the bit-shift with the numbers currently stored. If there is no argument, << initially holds 0 as the number of bits by which to shift.

list — In left inlet: The first number is bit-shifted to the left by the number of bits specified by the second number.

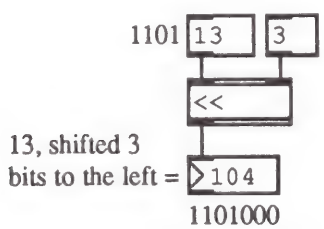
Arguments

int — Optional. Sets an initial value for the number of bits by which to shift leftward.

Output

int — The number in the left inlet is bit-shifted to the left by a certain number of bits. The number of bits by which to shift is specified by the number in the right inlet. The output is the resulting bit-shifted number.

Examples



Same effect as multiplying by a power of 2

See Also

*	Multiply two numbers, output the result
>>	Shift all bits to the right

Shift all bits to the right

>>

Input

int — In left inlet: All bits of the number, in binary form, are shifted to the right by a certain number of bits. The resulting number is sent out the outlet.

In right inlet: The number is stored as the number of bits to right-shift the number in the left inlet.

float — Converted to int.

bang — In left inlet: Performs the bit-shift with the numbers currently stored. If there is no argument, >> initially holds 0 as the number of bits by which to shift.

list — In left inlet: The first number is bit-shifted to the right by the number of bits specified by the second number.

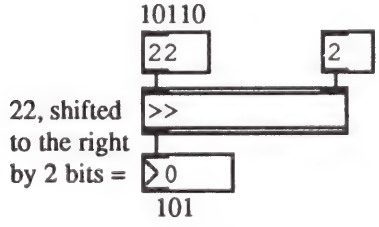
Arguments

int — Optional. Sets an initial value for the number of bits by which to shift rightward.

Output

int — The number in the left inlet is bit-shifted to the right by a certain number of bits. The number of bits by which to shift is specified by the number in the right inlet. The output is the resulting bit-shifted number.

Examples



Same effect as dividing by a power of 2

See Also

/ — Divide two numbers, output the result
<< — Shift all bits to the left

Index

abs 255
absolute value 255
accompaniment patch 130, 160
accum 93, 256
active 189, 257
active window 111, 232, 257
add, + 453
address
    of a funbuff 115
    of a table 420
aftertouch
    touchin 435
    touchout 437
aftertouch, polyphonic
    polyin 387
    polyout 389
algorithmic composition 247
Align 31, 230
All Notes Off message 74
All Windows Active 111, 232, 251
and 69, 464
animation 211
any window 232
append 107, 258
    received in a message object 108, 192, 343
Apple CD Rom driver 272
argument 13, 14, 36
    as a receiver of a message 343
    changeable argument 15, 24, 106, 192, 243, 343
    for specifying a MIDI channel 242
    for specifying a port 241
arithmetic operators 43
    % 457
    * 455
    + 453
    - 454
    / 456
array 195, 420
    funbuff 115, 308
    of symbols 196
    table 137
arrow icons in the PatchBay 241
ASCII 86, 323, 324, 365
Assistance 113, 233
    assigned to an inlet object 320
    assigned to an outlet object 367
associating a symbol with a number 278
Auto Fix Width 233
automatic actions 188
b 42, 261
background window 111
backslash 108, 192, 243, 343
bag 259
bang 15, 32, 223, 270, 345
    produced by clicking on a picture 441
    received in a table 144, 245, 421
    send to many places, in order 261
    time elapsed between 432
    when a window is closed 277
bangbang 17, 41, 261
beats per minute 134, 424
bendin 21, 72, 262
bendout 72, 264
bent patch cords 82, 233
bitwise and 466
bitwise operators
    & 466
    << 468
    >> 469
    | 467
bitwise or 467
bold type, displaying numbers in 363
Borax 21, 266
both numbers are not zero 69
Bucket 268
buddy 17, 269
bug
    debugging 197
    error message 204
button 32, 270
    as a debugging tool 198
    transparent 441
C function 182, 297
C programming language 4, 179, 216, 297
capture 149, 271
    debugging with 197
cd 272
change 70, 274
changeable argument 15, 24, 106, 192, 243, 343
channel mode message
    ctlin 286
    ctlout 288
    reference chart 237
channel pressure
    touchin 435
    touchout 437
characters, special 243
chord
    playing parallel chords 64
    storing 170
    transmitting 106, 127
chord-playing patch 170
clear the selection 227
clicktrack patch 190
clock icons in the PatchBay 241
clocker 132, 275
close the active window 226
closebang 188, 277

Index

coll 168, 196, 278
    data format 280
    editing the contents of 169
colored indicator, led 326
combining comparisons 69
comma 343
    in a mathematical function 243
    in a message box 106, 244
command
    from the Mac keyboard 86
command-period 249
comment 13, 38, 282
    clicking on 84, 441
commenting 198
comparison 67
    both numbers are not zero 464
    equal to 460
    greater than 463
    greater than or equal to 462
    if/then/else 318
    less than 458
    less than or equal to 459
    look for a number or symbol 401, 403
    look for a range of numbers 411
    look for a series of numbers 339
    not equal to 461
    one or both numbers are not zero 465
    report when numbers pass a threshold 371
    the greater of two numbers 340
    the greatest in a list of numbers 340
    the lesser of two numbers 352
    the smallest in a list of numbers 352
    using to make decisions 68, 76, 80
computational efficiency 199
conditional if/then/else statement 180
configuring MIDI 9
connecting MIDI equipment 8
constant value 199, 297
control change
    ctlin 73, 286
    ctlout 73, 288
    reference chart 237
controller numbers 73, 237
conversion of message type 199, 439
copy the selection 227
correctness checking 197
counter 132, 223, 283
creating a new object 238
ctlin 73, 286
ctlout 73, 288
cut the selection 227
cyclical pattern, creating 95
data byte
    of a MIDI message 235
    system exclusive 417
data structure 168, 195
    coll 196, 278
debugging 197
decimal number 14, 34
Decode 290
decrementing 94, 223, 283, 334
default value 37
del 97, 292
delay 97, 292
delaying
    a bang 97, 292
    before sending a note-off 337
    numbers 97, 383, 427
delta count 266
delta time 266, 356, 357, 432
dial 64, 293
dialog
    error 208
    Find... 228
    Get Info... 230, 253
    Midi Setup... 227
discrete values from a continuous stream 410
displaying messages 343
displaying numbers 363
divide, / 456
division of a beat 424
documenting patches 117, 203
documents 10
dollar sign 106, 192, 243, 343
drunk 295
dump banks to a synthesizer 221
dump patches to a synthesizer 217
duplicate the selection 228
duration
    reported by Borax 267
echo 97
Edit 230
Edit menu 227
Edit mode 29
editing a sequence 247, 357, 406
efficiency 199
encapsulation 202
entering numbers from the Mac keyboard 365
entering numbers in a Number box 363
equal to 460
error dialog 208
    stack overflow 134, 225
error message 198, 204
evaluate item text of a pop-up menu 341
exponential curve 182, 186
expr 179, 297
external clock
    synchronizing Max to 241, 275, 345, 402, 428
external object
    install 227, 240

Index

external objects 210, 211
    automatic loading 210
extra precision pitch bend data 155, 446, 447
fade-in, creating 132
File menu 226
file search path 25
filtering a stream of numbers 353, 410
filtering MIDI messages 21, 150, 199, 351
filtering out a specific number 72
filtering out note-off messages 412
filtering out repetitions
    of numbers 274
    of zero/non-zero status 433
Find Again 228
Find... 228
Fix Width 230
float 14, 34, 91, 299
floating point division 92
floating point variable 299
flush 61, 300
follow 159, 247, 302
font characteristics 39
Font menu 231
format
    of an mtr file 357
    of the script of a lib object 214
formatting
    MIDI messages 347
    numbers as text 426
formatting MIDI messages 150
fraction 14, 365, 456
frame 211
frequency distribution, histogram 144
fswap 17, 307
funbuff 115, 195, 308
function, mathematical 297
gate 77, 310
Get Info... 230, 251
    Edit menu commands 251
    table 138, 253
Ggate 76, 311
global variable 104, 445
grab 312
grace note patch 120
graphic 211, 313
graphics 211
graphics windows 211
greater than 463
greater than or equal to 462
grow bar 39, 84
Gswitch 76, 314
Help... 231
hexadecimal number
    displaying 51, 271, 363
    entering 244, 364
Hide On Lock 82, 231
Hide Subwindows 232
highlighting 441
Histo 144, 315
hslider 64, 316
if 180, 318
imitation 101
import a graphic 228
improvising patch 146
incrementing 94, 132, 152, 223, 283, 315, 334
inlet 12, 28
    Assistance description 113, 320
inlet object 24, 112, 320, 373
installation 7
installing external objects 227, 240
Install... 210
int 14, 34, 91, 321
integer variable 321
interface speed 226
inverting pitch values 66
invisible objects and patch cords 82, 231
IRCAM 4
iter 127, 322
key 86, 323
key code 323
keyboard commands 86, 323, 324
    entering numbers 88, 363, 365
keyboard slider 63
keyboard-like slider 325
keyup 89, 324
kslider 63, 325
labeling 282
least significant byte, pitch bend 446, 447
led 189, 326
left shift operator 468
less than or equal to 459
less than, < 458
lib 214, 327
libfrom 329
librarian 214
Librarian Window 251
library window 215, 327
libswitch 331
libto 332
limiting numbers to a specific range 65, 86, 316, 411, 443, 457
limiting the speed of a stream of numbers 74, 410
line 133, 334
    use with graphics 212
linear map of ranges, inverse 125
linear mappin of ranges 123
linking route or sel objects 207
list 14

combining numbers into 128, 170, 370, 427
convert to a series of numbers 127, 322
convert to separate numbers 127
in left inlet 45
separate numbers 442
load banks from a synthesizer 221
load patches from a synthesizer 217
loadbang 188, 336
loading a patch 199
send a bang when patch is loaded 336
locking and unlocking a Patcher window 29, 230
logical and 464
logical or 465
loop 223, 457
looping 94
main patch 24, 202
makenote 60, 337
mapping a range of numbers 123
masking 466, 467
match 339
Max documents 10
Max menu 230
Max Object List 240
Max Preferences 20, 232
Max window 232
printing in 394
maximum in a series of numbers 375
maximum number of presets 393
maximum object 340
memory usage 200
menu commands 226
menu object 172, 196, 341
message 14
append arguments at the end of 107, 258, 343
displaying 343
maximum length of 343
order of messages 16
prepend one before another 107, 391
reversing order of two numbers 128, 307, 415
routing to different destinations 310, 311, 401
viewing 197
message box 15, 106
message lookup 199
message object 28, 106, 343
$ argument 343
as a data structure 196
changeable argument 192
punctuation in 243
message type 199
metro 36, 345
MIDI
connecting MIDI equipment 8, 27
displaying note names 363
receiving and transmitting 349, 350
reference chart 236
sending and receving 21
MIDI channel
filtering by 151, 199
specifying 58, 242
MIDI Enable/Disable 21, 57, 189, 374
MIDI file 247
MIDI interface
connection 8
speed 226
MIDI Manager
configuration 241
time code conversion 241, 428
using 9
MIDI message 235
MIDI note name
displaying 51
MIDI note names
displaying 254
MIDI objects 21, 56, 235, 241
Midi Setup... 9, 226
MIDI time code 402
midiformat 150, 347
midiin 21, 149, 349
midiout 21, 149, 350
midiparse 21, 150, 351
Miller Puckette 4
millisecond 14
minimum in a series of numbers 440
minimum object 352
minus, - 454
modular programming 202
modulo 43, 457
Monk, Thelonius effect 120
mouse control 63
mouse status and location 185, 353, 354
mousefilter 184, 353
MouseState 185, 213, 354
mtr 164, 247, 356
multi-tracking 164, 356
multiplier, of a slider or dial 47
multiply, * 455
name of a send, receive, or value object 23
name of an object 12
negative numbers in a Table window 254
New menu 230
new object 238
New Object List 30, 232, 239, 250
adding names to 240
New Patcher 226
not equal to 461
note names, displaying 363
note-off message
filtering out 61, 412
holding, then outputting 413
supplying 60, 266, 300, 337, 385

Index

note-off with release velocity
    xnotein 449
    xnoteout 451
note-on/note-off indicator to xnoteout 451
notein 21, 57, 359
noteout 57, 361
number 14, 34
    displaying 363
    formatting as text 426
    send to many places, in order 439
    typing on the Mac keyboard 49, 88, 365
Number box 34, 49, 363
    as a debugging tool 197
numkey 88, 365
object box
    punctuation in 243
object palette 238
objects 11, 238
    aligning 31, 230
    creating 30, 238
    Max Object List 240
    MIDI 235, 241
    name of 12
    New Object List 239
    nonexistent 205
    size of, adjusting 39, 84
    storing settings of 392
    that evaluate expressions 297, 318
    using a patch as an object 24, 114
    width of 230, 233
octave
    parallel octave patch 57
    random octave transpositions 124
offset, of a slider or dial 47
omni mode 58
one or both numbers are not zero 69
onscreen control 63
open a subpatch window 373
open and close a subpatch window 374
Open As Text... 226
Open... 226
Options menu 232
or 69, 465
order of messages 16
ornamentation patch 120
outlet 12, 28
    Assistance description 113, 367
outlet object 24, 112, 367, 373
oval 211
Overdrive 95, 199, 232
    use with graphics 211
p 373
pack 128, 370
Page Setup... 227
pairing numbers 269, 370
palette of graphic editing tools 140, 252, 421
palette of objects 238
parameter of a synthesizer patch 214
    changing 332
PassPct patch 122
past 371
Paste 227
Paste Picture 83, 228
patch 17
patch cord
    receive messages without 104, 396
    segmented or straight 233
    send messages without 104, 405
    wiretap in 197
patch editor 214, 327
PatchBay 9, 226
patcher object 23, 111, 373
    argument to 193
Patcher window
    creating 226, 230
    locking and unlocking 230
    of a patcher object 373, 374
    of a subpatch 374
pcontrol 188, 374
Peak 375
periodicity 95
pgmin 72, 376
pgmout 72, 378
pics 380
PICS file format 380
pics2 380
pict 382
PICT file format 382
picture
    clicking on 84, 441
pipe 97, 383
pitch bend
    14-bit, xbendin 155, 446
    14-bit, xbendout 155, 447
    bendin 72, 262
    bendout 72, 264
    controlled with velocity 72
pitch-velocity grid 184
play a recorded sequence
    mtr 356
    seq 406
    with MIDI start message 402
plus, + 453
pointer 152
polling mouse status and location 354
poly 385
polyin 387
polyout 389

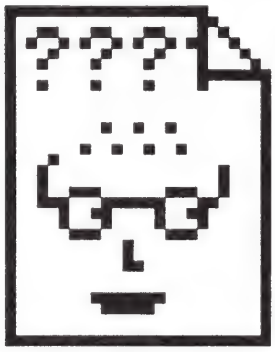
polyphonic key pressure
polyin 387
polyout 389
pop-up menu 341
port 241
specifying in Max 241
specifying in the PatchBay 241
when not using MIDI Manager 241
pound sign 243
prepend 107, 391
preset 175, 195, 392
print 15, 28, 394
as a debugging tool 197
Print... 227
probability 119, 144, 245, 315, 395
program change
pgmin 72, 376
pgmout 72, 378
protocol of MIDI messages 235
punctuation
in a message object 243
in an object 243
quantile 245, 315, 421
quit Max 227
r 104, 396
ramp of number values 334
random 99, 395
random note patch 99
random number
drunk 295
weighted randomness 245
range
of a slider 46
real time 4
rtin 402
receive 23, 104, 396
double-clicking on 250
receiving MIDI 21, 349
recorder patch 164
recording a sequence of messages 356
recording a sequence of MIDI messages 406
rect 211, 397
recursion 117, 134
relational operators 67
!= 461
< 458
<= 459
== 460
> 463
>= 462
remainder 457
repeat actions 223, 345
repeated note patch 93, 176
Replace 229
Resume 208, 231
rhythm
analyzing 145
creating with delay 99, 130
right shift operator 469
right-to-left 16
switching the order 261, 307, 415, 439
ring 211, 399
ritard, following a performer 162
route 78, 401
linking route objects 207
routing
a range of numbers 411
messages to different destinations 68, 76, 86, 310, 311, 401
rtin 402
s 104, 405
save a file as text 226
save a Max document 226
Save As... 226
scheduler 208
score-reading object 159, 302
script of a lib object 214, 327
example script 222
search path 25
searching a CD 272
Segmented Patch Cords 82, 233
cancelling 250
sel 403
linking sel objects 207
select 67, 403
select all objects in the active window 228
selector 14
semicolon 243, 343
in a message 109, 244
send 23, 104, 405
double-clicking on 250
Send to Back 232
seq 157, 247, 406
sequencing 157, 247
editing a sequence 357, 406
multi-track 164, 356
saving a sequence 158, 357, 406
single track 157, 406
serial port 8
set
received in a message object 107, 192, 343
received in a slider 47
settings of objects, storing 175, 392
sharp sign 118
shift operator 468, 469
Show On Lock 83, 231
signed values in a table 254
sine wave, drawing 182
size
of a loaded patch 199

Index

of a table 253
of objects, grow bar 39
slider 46, 408
multiplier 47
offset 47
setting the range of 46
SMPTE conversion 428, 430
special character 243, 343
speed of computation 199
speedlim 74, 410
split 86, 411
stack overflow 134, 208, 225
Startup Directory 227
status byte
system exclusive 417
stealing voices 385
stopwatch patch 188
storing
messages 278
settings of objects 175, 392
stripnote 61, 412
stuck notes, avoiding 60, 266, 300, 337, 385
subpatch 23, 111, 114, 373
argument to 118, 193
opening the window of 188, 373, 374
subtract 454
subwindow 232
sustain 413
swap 17, 128, 415
switch 77, 416
sxformat 153, 417
symbol 15
received in a message object 192, 343
synchronizing 241, 275, 402
synchronous messages 269
synthesizer
changing patch parameters 332
load or dump a bank of sounds 221
load or dump patches 217
patch editor/librarian 214
switching sounds 73, 174, 378
sysexin 419
system exclusive 153
end byte 154, 417
status byte 154, 417
sysexin 419
system exclusive programming 154, 214, 417
t 42, 439
table 137, 195, 420
Don't Save 138, 422
editing window 252
entering values as text 139, 151, 230, 421, 426
quantile message 245
Save with Patcher 138, 422
saving in a file 138, 421
size of 420
stepping through values 152
storing numbers in 140
using a saved table file 254
viewing 137, 252
Table window 230, 252
creating 139, 252
editing 140
Select All 228
shortcuts 251
tempo 134, 424
testing a patch 197
Text 426
Text window 230
The Max Window 11
thresh 170, 427
threshold
numbers received within 427
when numbers go beyond 371
time code conversion 241, 402, 428, 430
time elapsed between events 89, 266, 275, 432
timed progression of numbers 132
timed repetition 36, 224, 345
timein 428
timeout 430
timer 89, 432
times, * 455
title bar 21, 230
TogEdge 147, 433
toggle 38, 434
toggling 290, 314, 326, 433, 434
touchin 435
touchout 437
transmitting MIDI 21, 350
transparent button 441
transposed note, turning off 87, 114, 159
transposing 57, 114
trap a number or symbol 403
trigger 17, 42, 439
triggering 16, 43
Trough 440
truncation 14, 34
type of message 199
conversion 439
ubutton 84, 441
undo the most recent editing change 227
unpack 127, 442
Use Apple MIDI Manager 9, 226
user interface object 13
uslider 64, 443
v 104
value 104, 445
double-clicking on 250

variable 91
variable, global 104, 445
voice allocation
    Borax 266
    poly 385
voice stealing 385
weighted randomness 119
width of an object 230, 233
wild card 339
Windows menu 232
xbendin 446
xbendin2 446
xbendout 447
xnotein 449
xnoteout 451
zoom the view of a table in or out 254
!= 67, 461
$
    in a message box 106, 192, 343
    in an object box 193, 297, 318
% 43, 457
& 466
&& 69, 464
* 43, 455
+ 43, 453
- 43, 454
/ 43, 456
0x, hexadecimal indicator 244
< 67, 458
<< 468
<= 67, 459
== 67, 460
> 67, 463
>= 67, 462
>> 469
\ 108, 192, 343
| 467
|| 69, 465

Writing External Objects for Max



Revision 4 (Little Shark) of 31 August 1990

written by David Zicarelli

Opcode Systems, Inc.
3641 Haven Ave., Suite A
Menlo Park, CA 94025

This document © 1990 Opcode Systems, Inc.
Max © 1990 IRCAM

Table of Contents

1. Overview 4
    Conventions 5
2. What You Need to Do in Think C 6
3. Defining Your Object 8
    Function Macros 8
    About Initialization Time 9
    Setting up your Class 9
    Identifying Your Object 10
    External Objects and Resources 11
4. About Type Lists and Atoms 13
5. Messages 16
    Expected Behavior 16
    Common Messages 17
    Messages from the System 17
    Routines for Adding Messages 20
6. Your Object Creation Function 22
    Inlets and Outlets 23
    Routines for Creating Inlets 23
    Routines for Creating Outlets 24
    Proxies 25
7. Writing Methods 27
    Routines for Using Outlets 27
    Binbufs and Max Files 28
    Binbuf Routines 30
    Clocks 32
    Qelems 35
    Essential Max Utilities 37
    Miscellaneous Operating System Utilities 40
    Routines for Dealing with Files 40
8. Advanced Facilities 42
    Advanced Object Creation and Message Routines 42
    Table Access 43
    Text Editor Windows 44
    Access to expr Objects 46
    Presets 47
    Event Serial Numbers 48
    MIDI Manager Access 49
9. Doing Windows 51
    Basic Window Messages 52
    Advanced Window Messages 53
    Menu Messages 54
    Window Routines 57
    Numericals 60
10. Writing User Interface Objects 62
    Boxes 62
    The All-Important SICN 63
    User Interface Object Creation Function 63
    User Interface Object Free Function 65
    Additional Messages for User Interface Objects 66
    Miscellaneous User Interface and Patcher Routines 68
    User Interface Objects with Text 71

11. Graphics Windows 73
Graphics Window Routines 73
Offscreen Routines 74
Sprite Routines 75
A Sprite Example 77
12. Writing External Functions for expr 81
What To Do In Think C 81
Types and Declarations 82
Initialization 83
Performing Computation 83
13. Index of Function Macros 86

1. Overview

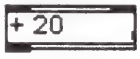
Writing External Objects for Max gives you an inside peek at Max and shows how the environment can be extended through code resources. This document assumes some familiarity with Max from a user's standpoint. However, we'll try to show the connection between the programming constructs presented and the user's experience of them.

Writing external objects for Max is somewhat analogous to writing a Hypercard XCMD. Your task is to write a code resource that gets loaded and called by the "master environment" and in turns calls upon helpful routines back in the master environment.

However, in Max, you create a *class*, or template for the behavior of an object. Instances of this class actually "do the work" of the object, when they are sent messages. Your external object definition will:

- Define the class: its data structure, size, how it is to be created and destroyed
- Define functions (called *methods*) that will respond to various messages, performing some action

There are two kinds of objects you can write. *Normal Objects* show up in Patcher windows in New Object boxes with two lines at the top and bottom, like this:



User Interface Objects are a bit more complicated to write, but they can have any appearance and behavior in a Patcher window, such as this **hslider**:

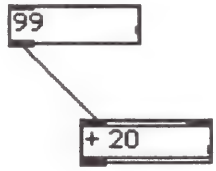
Typically, if you'll be interfacing Max to the "outside world" or performing some computation, you'll write a normal object. Normal objects can also open their own windows and dialog boxes. But they can't do any drawing or event handling in the Patcher window itself. To do anything earth-shaking within a Patcher window, you'll need to do some "extra work" to make a user interface object.

As mentioned above, there are two phases of your object definition: class initialization time and, for lack of a better phrase, object behavior time. Your code resource can be loaded when Max starts up if it's placed in Max's startup folder (usually called *max-startup*), or it can be loaded whenever Max wants to create the first instance of your object. At this time your object's main function will be called, and it should initialize its class. After the class is initialized, your routines will not be referenced until someone creates an object (instance) of your class. This happens...

- when a file is read in
- when someone types the name of your object into a New Object box in a Patcher window
- when duplicating an existing object of your class

Your creation function will be run at this point. In this routine, you'll allocate memory for a new object of your class, and do additional initialization for the fields of the object.

After the object has been created, it will receive messages. When a number is sent in an object's inlet, the object receives an int message (or a float message, if it's a floating point number).

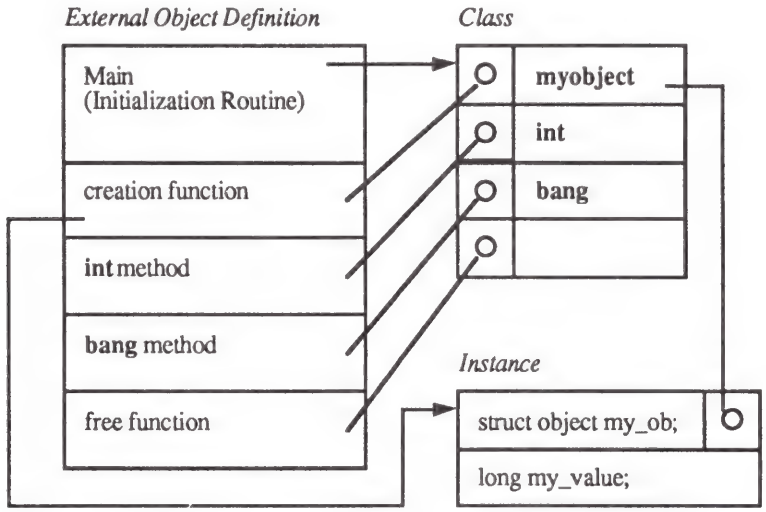


You'll write a method to respond to this message. If you were an object that performed addition, your int method might add two numbers together and send the result out your outlet.

There are a number of different predefined messages you can respond to. You can even define new messages yourself. Defining messages and associating methods with them is done at initialization time when you're setting up your class.

Finally, if your object is deleted, your free function will be called. If you didn't allocate any extra memory inside your object (assigned to any of your object's fields), you need not have a free function. Otherwise, you should free the memory of these fields in this function.

The figure below shows the relationship between the external object definition you write, the class, and an instance.



Conventions

Macintosh Traps are printed in Courier with a preceding underscore (`_TrapName`). The names of Max messages are in Helvetica Condensed (message) and are always all lowercase. Names of existing (built-in) Max objects are in bold Hevetica Contensed (**object**). Other Max programming names and constructs (`wind_drag`) are in Courier. Menu commands in Max and other programs are capitalized in bold. Any messages you might see in the Max window will be printed in Courier italic (*`Method not found`*).

2. What You Need to Do in Think C

The first thing you need to create an external object code resource in Think C is a special version of the MacTraps library which doesn't contain the Quickdraw globals. To do this...

- Make a copy of the MacTraps file in the Finder, call the copy **MacTraps w/o Quickdraw**, and open it as a project in Think C.

- Remove the item named **QDGlobals.a.π**. Close the project.

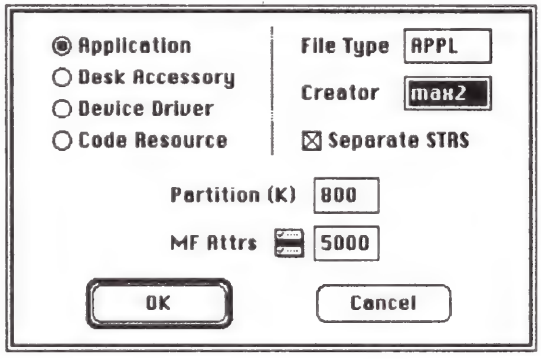
You're now ready to create a new project file for your code resource.

- Click on the **New** button in the open project dialog box, or choose **New Project...** from the Project menu. Name the new project.

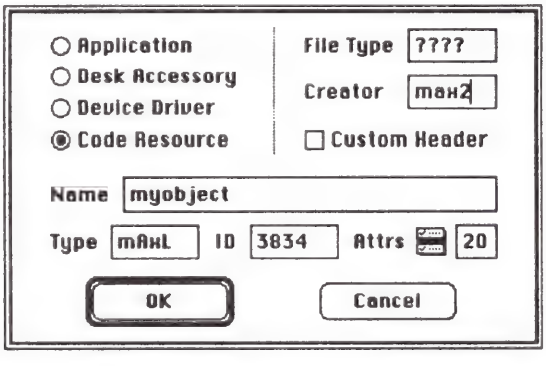
- Add the **MacTraps w/o Quickdraw** file to the project by choosing **Add...** from the Source menu.

Your external routine will be written as a code resource. In order to tell the compiler what you're doing, choose **Set Project Type...** from the Project menu in Think C.

You'll see the following dialog box.



- Change it to look like the one below.



- Choose a number for the **mAxL** resource. The particular number doesn't matter because only one such code resource will be open at the same time.

- Close the dialog box.

- Open a new text file for the C code for your code resource, and write your code.

A number of necessary C header files are provided in the folder **#MAX includes**. Your C files always need to include the file **ext.h**. If you'll be writing user interface object, include **ext_user.h**. If you'll be opening a window, include **ext_wind.h**. These files should be included after normal Macintosh include files because they use Macintosh type definitions like GrafPort and so forth. Because you'll need to access "globals" in your code resource, you'll need to include the Think C include file **SetupA4.h**.

We haven't included type definitions for many of the Max structures we talk about below (Binbuf, Clock, Qelem, etc.) because you won't need to know their internal structure to use them. Typically, you can just refer to them by pointers to void (`void *`).

3. Defining Your Object

You need to define a structure for your object. If you're defining a normal object, it needs to start with an structure called an Object. This is not a pointer, but an entire Object structure contained inside your object. Miller Puckette's convention (after Kernighan and Ritchie?) in writing Max was to start fields of data structures with a lowercase letter followed by an underscore. Usually the letter is the first letter of the name of the structure. Here's a hypothetical structure for an **Alarmclock** object.

```
typedef struct alarmclock {
    Object a_ob;
        /* must be the first thing in any non user-interface object */
    long a_hours;
    long a_minutes;
    long a_seconds;
    long a_alarmset;
} Alarmclock;
```

The Object contains references to your object's class definition as well as some other information. This class reference allows an instance of your class to respond to messages in the right way.

Traditionally, fields inside of Max objects are declared as longs, but this only because all integers in Max are longs. You're free to use any size object you wish. Keep in mind that Max passes floating point numbers as type `float`, so any extra precision not contained in a `float` will be lost. This doesn't mean you can't perform extra-precision computation inside your object.

Function Macros

Max provides you with a sizeable number of functions to assist you in writing external objects. These are typically the same functions that the objects built into Max use to carry out their work. Since your code resource cannot know the addresses of these functions because it isn't linked into Max itself, Max hands your `main` function a pointer to a table of functions. **ext.h** (found in the **MAX #includes** folder) contains macros that access this table. The macros present a programming interface substantially identical to that you'd have if you were writing the object in Max itself. In order to access the macros as declared in **ext.h**, you'll need a global variable declared as follows:

```
fptr *FNS;
```

One of the very first things you do is assign the pointer to the function table you were passed in `main` to `FNS`. After `FNS` is assigned, you can access the function macros within `main` and all other functions you write in your object definition.

Note: Hereafter we'll refer to Max's function macros as simply *functions*.

About Initialization Time

Your code resource contains only one entry point known to Max—the starting address of your function `main`. Your `main` function is called once and only once—when the code resource is loaded. It should look like this:

```
fptr *FNS;
    /* holds Max function macros table, fptr declared in ext.h */

void *class;    /* points to your class */

void *myCreationFunc(), *myFreeFunc(), *myMenuFunc();

main(f)
fptr *f;
{
    RememberA0();
        /* A0 holds pointer to your code resource's globals */
    SetupA4();
        /* do this for every function called from Max that accesses
            your globals */

    FNS = f;            /* now all the function macros will work */

    setup(&class, myCreationFunc, myFreeFunc,
        sizeof(MyType), myMenuFunc, 0);
        /* allocates class memory and sets up class */

    /* add messages here using addmess, addint, addbang, etc. */
    /* copy any resources here using rescopy */

    RestoreA4();    /* counterpart to SetupA4() */
}
```

Setting up your Class

```
setup (void **class, methodptr createfun, methodptr freefun,
      int classSize, methodptr menufun, int types…)
```

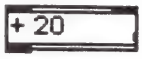
`setup` is the first function your external routine should call at initialization time (after setting up register A4 to point to your globals). It creates a class definition for you, which you'll then use when creating new objects of your class.

`class` should be a global variable in your code resource, which can be declared as

```
void *class;
```

The first argument you pass to `setup` should be a pointer to this global. `createfun` and `freefun` are pointers to your creation and free functions. If you don't need to do any special cleanup (the memory occupied by an object and all its inlets and outlets are automatically freed when it goes away), you can pass `0L` for `freefun`. `classSize` should be the size of your object's data structure in bytes, so that memory for an instance can be allocated properly. `menufun` is used if you're building a user interface object. It's the function that gets called when the user creates a new object of your class from the Patcher window's palette. Pass `0L` if you're not defining a user interface

object (we'll discuss `menufun` in the section on Writing User Interface Objects). `types` is a standard Max type list as explained below in the section called About Type Lists and Atoms. The final argument of the type list should be a 0. This list specifies the arguments which are expected when a new instance of your class is created. These would be the arguments that the user types in after the name of your class. For example, if the user types...



...into an object box, 20 is the argument passed to the + class's instance creation function.

The hypothetical class `myobj` is created using `setup` below.

```
/* the data structure for the object */
typedef struct myobj {
    struct object m_ob;
    long m_watchtower;
} Myobj;

/* the creation and free functions */
void *myobj_new(),*myobj_free();

/* the global variable that points to the class */
void *class;

main(f)
fptr *f;
{
    RememberA0();
    SetupA4();
    FNS = f;
    setup (&class,myobj_new,myobj_free,
        sizeof(Myobj),0L,A_DEFLONG,0);

    /* additional code will go here */

    RestoreA4();
}
```

After calling `setup`, you'll have a perfectly well-defined class that doesn't know how to do anything. In order to make it useful, you need to make the class respond to messages. "Respond to a message" means that a symbol (such as the word bang) is bound to a function you write (called a method). We'll discuss the functions you'll use to do this in a moment. These functions are designed to make it easy to add standard messages to your class. `addmess`, however, allows you to specify novel messages and give them arguments which will be type-checked for you by Max. About Type Lists and Atoms discusses how to specify and type-check arguments.

Identifying Your Object

```
alias (char *otherClassname)
```

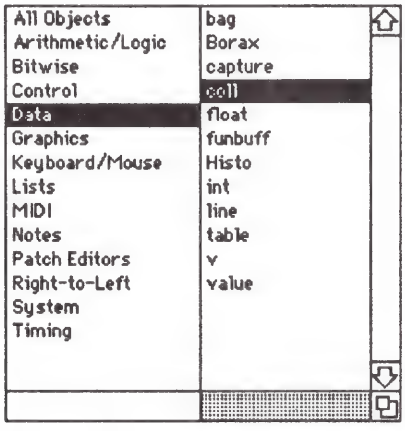
This function allows users to refer to your object by a name other than the the name of the mAxL code resource. This might come in handy if you're writing an external object which could have a number of possible manifestations, such as a shape drawing object which could easily do ovals or

rectangles. If you call `setup` with a type list of `A_GIMME` (this is explained below) your object creation function will be able to find out the name the user typed to create the object.

However, it's not so simple! If the user wants to load your file dynamically when typing the name of your object, the filename has to be the same as the name of the object. Thus, if you use `alias`, you should inform the user that your object needs to be loaded at startup (or by choosing **Install...** from the File menu), otherwise the aliased names will result in "no such object" errors when they are first referenced.

```
finder_addclass (char *category, char *classname)
```

This function adds your object's class name to New Object List that appears when the user clicks to add a "new object" in a Patcher window. The categories are listed on the left side of the window.



If you pass `0L` for `category`, your `classname` will be added only to the "All Objects" list. Refer to a recent version of Max for a list of current categories. A category name which is not among the current categories will not be created for you; instead, your object will only be listed under "All Objects."

You may call `finder_addclass` as many times as you like, either to install the name of your object under a number of different categories (try to restrain yourself), or to install alias names for your object. Here's what would be necessary to install an object called **oval** (and its alias **rect**) in the *Graphics* category.

```
alias("rect");
finder_addclass("Graphics","oval");
finder_addclass("Graphics","rect");
```

External Objects and Resources

The file which contains your code resource and any other resources is closed after initialization time. That means that if you wish to use any Macintosh resources such as dialogs or menus inside your object's methods, you'll need to copy them out of your code resource file. `rescopy` is a function which copies a resource you specify from your original file to a temporary file. You may have seen this file (called *Max Temp 1*) when running Max in MultiFinder or after has Max crashed.

```
rescopy (OSType resType, int resID)
```

Designed to be used at initialization time, `rescopy` copies the resource of the specified type and ID from your code resource file to a Max temporary file. It should work even when running Max on locked media. After the resource have been copied, you can access them as if they were part of your file. `rescopy` returns 0 if successful, otherwise an error will appear in the Max window and `rescopy` will return -1.

Resources other than your **mAxL** code resource have to be numbered in such a way as to avoid conflicts with Max's own resources and those of other external objects. If you're curious about Max's resource IDs for a given type, just look at the Max application file in ResEdit. We may devise a way to assign resource IDs to particular individuals in the future. For now, just try to use obscure large (i.e. > 1000) numbers not easily divisible by anything.

4. About Type Lists and Atoms

Max can provide you with the service of type checking arguments to messages. The two functions setup and addmess take argument type specification lists.

Your object creation function (as passed to `setup`) is actually called in response to a message sent to a special **new** object. You pass the same kind of argument type specification list to setup as you do when defining the arguments to your own messages. In the case of a message to the **new** object, the name of the class is the "message" itself, and the arguments are what follows the class name (the 20 in + 20, for example).

The function `addmess`, like `setup`, takes argument type specification lists. For example, suppose we want to define a message search which requires two long integers as arguments. The user might type such a message into a Max message box connected to our object.

In this case, search is the message, and 304 and 228 are the arguments. The argument type list would look like this:

```
A_LONG, A_LONG, 0
```

Argument type lists always end with a zero (defined as `A_NOTHING`).

You would declare the arguments to the function (method) you write to respond to the search message as follows:

```
void *mySearchMethod(x, arg1, arg2)
myObjectType *x;
long arg1, arg2;
{

}
```

Other types you can specify are floating point numbers with `A_FLOAT`, and Symbols (text words) with `A_SYM`.

If you want arguments to be optional, you can use `A_DEFLONG`, `A_DEFFLOAT`, and `A_DEFSYM`. These default missing arguments to 0, 0.0, and the empty symbol ("") respectively.

In each case where you declare explicit arguments, Max will pass arguments of the specified type directly to your method. If the arguments are of the wrong type, Max will indicate an error to the user.

To review, here are the basic types you'll work with:

```
#define A_NOTHING       0
#define A_LONG          1
#define A_FLOAT         2
#define A_SYM           3
#define A_OBJ           4
#define A_DEFLONG       5
#define A_DEFFLOAT      6
#define A_DEFSYM        7
```

You can only specify up to seven arguments in a list. However, you can specify that Max just hand you the arguments as an array of *Atoms* (a structured type defined below) if you use the following type list:

```
A_GIMME, 0
```

This allows you to type check the arguments yourself, and no limit is placed on the number of arguments which can be included in such a message. A_GIMME is defined as follows:

```
#define A_GIMME         8
```

An Atom has the following form:

```
union word              /* union for packing any data type */
{
    long w_long;
    float w_float;
    Symbol *w_sym;
    Object *w_obj;
};

typedef struct atom             /* an Atom which is a typed datum */
{
    int a_type;         /* from the definitions above*/
    union word a_w;
} Atom;
```

If you declare a method to receive its arguments with A_GIMME, they'll be passed as an *argc, argv list* (familiar to UNIX C programmers). argc is the number of arguments, and argv points to the first of argc Atoms in an array. You're also passed the Symbol containing the message itself. If your creation function gets its arguments a la A_GIMME, this Symbol is the name of your class. Here's an example function declaration that corresponds to an A_GIMME type list.

```
void *myMethod(x, s, argc, argv)
myObjectType *x;
Symbol *s;
int argc;
Atom *argv;
{

}
```

To type-check the arguments yourself, you look at the a_type field of an Atom. The possible values are A_LONG, A_SYM, and A_FLOAT. Here's an example method that prints out its arguments that could be used as a model for type-checking and processing arguments:

```
void *myMethod(x, s, argc, argv)
myObjectType *x;
Symbol *s;
int argc;
Atom *argv;
{
    register int i;

    SetUpA4();
    for (i=0; i < argc; i++) {
        switch (argv[i].a_type) {
            case A_LONG:
                post("argument %ld is a long: %ld", (long)i,
                    argv[i].a_w.w_long);
                break;
            case A_SYM:
                post("argument %ld is a symbol: name %s", (long)i,
                    argv[i].a_w.w_sym.s_name);
                break;
            case A_FLOAT:
                post("argument %ld is a float: %lf", (long)i,
                    argv[i].a_w.w_float);
                break;
        }
    }
    RestoreA4();
}
```

The Symbol argument to your method is the name of the message which invoked your method. You may have several symbols bound to the same method. For example:

```
addmess(myObject_doit,"doit", A_GIMME, 0);   /* bound to doit */
addmess(myObject_doit,"finger", A_GIMME, 0); /* bound to finger */
```

If you want to know when your method was invoked with the doit message, check to see if the Symbol `s` is equal to the doit symbol, using the following technique:

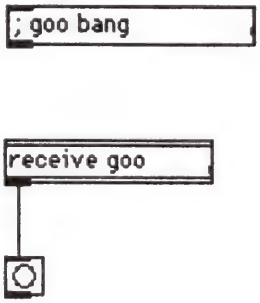
```
if (s == gensym("doit")) {
    post("wow, I was called by doit");
}
```

Note: `gensym` is a function which returns the Symbol associated with a character string.

5. Messages

Your object will do its work by *responding to messages*. Objects commonly receive numbers (int and float messages) in their inlets, process them, and send them out their outlets. However, arbitrary messages can be sent to objects when a Max user uses the message box.

The Max *message box* sends any message after a semicolon to the first symbol, called the *receiver*. For example, the patch below shows the **receive** goo object can receive a bang message (and light up the button) without actually being connected to the message box that's sending it.



Anyway, the point is that you have a very extensible "language" at your disposal, and it's pretty wide open to invent a "command structure" for your object. There are a few guidelines we expect external object writers to adhere to.

Expected Behavior

The most basic thing your object will do with a number should be in the method that responds to an integer in the left inlet (the int message). You should store this number somewhere inside your object too, so that if the object receives a bang message, it repeats the action performed by sending the integer, using the stored value.

You'll get a different message when a number is sent in an inlet other than the leftmost one. The leftmost inlet gives you an int message whereas other inlets give you in1, in2 etc. (in1 is associated with the inlet next to the leftmost one, and the in-number increases as you move to the right).

In most cases, you'll want the leftmost inlet to cause the object to output value or perform some kind of action, while the other inlets are used to store additional information needed when the action is taken.

As an example, consider the **noteout** object. Its leftmost inlet specifies the pitch of the note to be played, while the other inlets set the MIDI channel and velocity. The internal structure of **noteout** looks something like this:

```
typedef struct noteout {
    Object n_ob;
    int pitch, velocity, channel;
} Noteout;
```

If the **noteout** object had three methods, one corresponding to each of its three inlets, they would behave as follows:

int	set pitch, send MIDI note message using current velocity and MIDI channel
in1	set velocity
in2	set MIDI channel

By the way, if you send a list of three integers to an object such as **noteout** that has three inlets, Max will automatically separate the list into the three individual messages—the third number will be sent as an in2 message first, then the second number will be sent as in1, and finally the first number will be sent as int. This behavior will not occur if your object has a method that responds to a list message (see the discussion of `inlet_new` below for more information about list methods).]

After writing our three integer methods, we'd also want to add a bang method, which sent a MIDI note message using the current values of pitch, velocity, and MIDI channel stored in the object.

Adding many ways to accomplish the same task adds flexibility to how your object can be used. For example, Max would be a lot harder to use if **noteout** *required* a list of three numbers every time you wanted to send a note. On the other hand, allowing people to play notes by sending a list of three numbers might help someone accomplish what they want to do more easily.

Common Messages

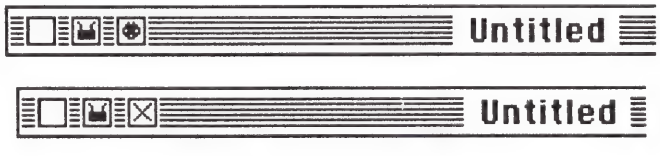
If you're planning to use a specific word as a message, try to use a word that might already be in use in existing Max objects. These words include:

set N	Sets a value without causing output
start	Starts something, or use bang
stop	Stops something
record	
append	
read S	Read a file (also use import), S is an optional name
write S	Write a file (also use export), S is an optional name
next	Output the next value, go to the next thing
prev	Output the previous value, go the previous thing
goto N	Set the next or prev counters to N
size N	Set size to N
zero	Set to all zeros
clear	Delete, erase, set to zero
length	Output your length

Messages from the System

There are some predefined messages you may want to respond to:

The enable and disable messages are sent when the user clicks on the MIDI icon in a Patcher window title bar.



When turning the MIDI icon into an X, the disable message is sent to all the objects in the window. Of existing Max objects, only MIDI objects (i.e. **notein**, **noteout**) respond to this message, but if your object communicates directly with the outside world in a manner analogous to MIDI, you might consider disabling communication in response to a disable message and re-enabling it in response to an enable message. Always create your object in an enabled state because you'll never receive an initial enable message.

The info message is sent to your object if it's selected in a Patcher window and the user chooses **Get Info...** from the Max menu. This might be a time to put up a dialog box to set or display your object's parameters. You might also take this opportunity to put up an About box to brag about how great you are for being able to write a Max external object. At initialization time, add an info method with:

```
addmess(myobject_info,"info",A_CANT, 0);
```

Declare the method as follows:

```
void *myobject_info(x,p,b)
Myobject *x;
Patcher *p;
Box *b;
{

}
```

The definitions of Patcher and Box are in **ext_user.h**. You may not need to access your Patcher or your Box, in which case you can declare your info method as:

```
void *myobject_info(x,p,b)
Myobject *x;
void *p;
void *b;
{

}
```

You respond to a preset message by supplying a message which will restore your object to its current state. This may be done with a set or int message, or something more complex if your object has a lot of different data that can be changed. At initialization time, add the preset method with:

```
addmess(myobject_preset,"preset",0);
```

Declare the method as follows:

```
void *myobject_preset(x)
Myobject *x;
{

}
```

The functions `preset_int`, `preset_set`, and `preset_store` are useful in writing your preset method. See the section on Presets below for descriptions of these routines.

When a Patcher window is loaded, each object it contains receives the loadbang message. There is a built-in Max object called **loadbang** which sends out a bang when a Patcher window is loaded. Your object can respond to the loadbang message in any way it wants. Before implementing a method that responds to loadbang, keep in mind that the user can connect your object to the outlet of a **loadbang** object to perform some kind of initialization if necessary. Note that you do not get the loadbang message when the user creates a new instance of your object in the Patcher window, only when a Max file containing your object is read from disk.

At initialization time, you add a loadbang method with:

```
addmess(myobject_loadbang,"loadbang",0);
```

Declare the method as follows:

```
void *myobject_loadbang(x)
Myobject *x;
{

}
```

The assist message is sent to your object when the user has positioned the cursor over one of your object's inlets or outlets and the Assistance area of the Patcher window is visible. To respond to the assist message, you should provide a description of the inlet or outlet's *raison d'etre* in 60 characters or less. See example Assistance messages from built-in objects. You can even get fancy and refer to specific arguments or the state of the instance, although there's no guarantee you'll be sent another assist message if your state were to change.

At initialization time, you add an assist method with:

```
addmess(myobject_assist,"assist",A_CANT,0);
```

Declare the method as follows:

```
void *myobject_assist(x,b,msg,arg,dstString)
MyObject *x;
void *b;
long msg,arg;
char *dstString;
{


}
```

`b` is a pointer to your object's box. `msg` is one of two values:

```
#define ASSIST_INLET 1
#define ASSIST_OUTLET 2
```

You should ignore `msg` values you don't know about. Future versions of Max may ask for additional kinds of assistance from you.

`arg` is the inlet or outlet number, starting at 0 for the leftmost inlet or outlet. `dstString` is where you should copy a C string with the Assistance information. To make your object as internationalizable as possible, you should try to store your Assistance strings as resources, but if you do it the quick and dirty way, we understand that there are only so many hours in a day.

Here's a shameless example done without resources for an object with three inlets and one outlet, just to give you the idea of what you're to do.

```
void *myobject_assist(x,b,msg,arg,dstString)
MyObject *x;
void *b;
long msg,arg;
char *dstString;
{
    SetUpA4();
    if (msg==ASSIST_INLET) {
        switch (arg) {
            case 0:
                strcpy(dstString,"Start Clock Ticking");
                break;
            case 1:
                strcpy(dstString,"Set Clock Speed in Milliseconds");
                break;
            case 2:
                strcpy(dstString,"Set Clock Erratic Factor (0-99)");
                break;
        }
    } else if (msg==ASSIST_OUTLET) {
        strcpy(dstString,"Clock Ticks");
    }
    RestoreA4();
}
```

See the utility function `assist_string` which helps you simplify the use of 'STR#' resources in handling assist messages.

Routines for Adding Messages

`addbang (methodptr mp)`

Adds the function pointed to by `mp` to respond to the bang message.

`addfloat (methodptr mp)`

Adds the function pointed to by `mp` to respond to a float in the leftmost inlet.

`addftx (methodptr mp; int n)`

Adds the function pointed to by `mp` to respond to a float in the inlet `n` inlets to the right of the leftmost inlet. For example, if an object has two inlets, `addftx(mp,1)` would add the method `mp` to respond to a float in the rightmost inlet.

`addint (methodptr mp)`

Adds the function pointed to by `mp` to respond to a long integer in the leftmost inlet.

```
addinx (methodptr mp; int n)
```

Adds the function pointed to by `mp` to respond to a long integer in the inlet `n` inlets to the right of the leftmost input. For example, if an object has two inlets, `addinx(mp,1)` would add a method to respond to an integer in the rightmost inlet. Note: This correspondence between inlet locations and messages is not automatic, but it is strongly suggested that you follow existing practice. You must set the correspondence up when creating an object of your class with proper use of `intin` and `floatin` in your creation function (see Your Object Creation Function).

```
addmess (methodptr mp; char *s; int types ...)
```

Adds the function pointed to by `mp` to respond to the message string `s` in the leftmost input. Type checking of the message's arguments can be done by passing a list of argument type specifiers (described under About Type Lists and Atoms). The list must end with a 0. The maximum number of type-checked arguments is 7.

6. Your Object Creation Function

As with all functions you write that will be called by Max, you'll need to know what the arguments to your object creation function will be based on an argument type list you specified. Recall that the types for the creation function's arguments were passed to `setup` at initialization time.

The creation function is called identically to the methods described above under About Type Lists and Atoms with one exception. Since the message that's being responded to was not sent to an object of your class, but to the special **new** object, the first argument will not be a pointer to an object of your class. For example, if your creation function's argument type list were...

```
A_DEFLONG, 0
```

your creation function should be declared as

```
void *myCreationFunc(arg)
long arg;
{

}
```

The task of your creation function is to create an object which is an instance of your class. The creation function should return a pointer to the created object or 0 if there was a problem.

As with all functions called from Max, the first thing you do in your creation function is call `_SetupA4`. Then you'll want to call `newobject` to create the object. `newobject` does all the work of making your object an upstanding member of your class. Then you'll perform additional initialization of your object's fields. For example, you might want to add an Outlet, a Clock (so the object can schedule itself), or a Qelem (if the object will draw anything or perform any action which cannot take place at interrupt level). Some fields could be assigned values based on the arguments you were passed. When you type...

...the **+** object's creation function takes its argument 20 and stores it inside the newly created object.

Note: If you want to access your object's Box structure in the Patcher, you must first do so in your object creation function. The technique for doing this is shown in the description of the function `patcher_dirty`.

```
void *newobject (void *class)
```

You call `newobject` when creating an instance of your class in your creation function. `newobject` should be passed the global variable which points to your class. This global pointer was set by the `setup` function you called at initialization time. `newobject` allocates the proper amount of memory for an object of your class and installs a pointer to your class in the object, so that it can respond with your class's methods if it receives a message.

Inlets and Outlets

Inlets and *Outlets* are the way your object normally communicates with other objects. They are structures which keep track of connections between objects and facilitate sending messages "through" these connections.

When your object is created, you're usually given an Inlet. It is referenced in the Object structure that should be the first field of your object. Notice that some objects, such as **notein**, don't show inlets. This is because they set a special flag in their class field that says, "Don't give me an inlet." You can set this flag yourself at initialization time. Here's how:

```
Messlist *mp;
Class *cp;

mp = class;     /* assign variable of type Messlist to point to your
                   global class variable */
cp = mp-1;  /* the Class is stored as the -1th element of your Messlist */
cp->c_noinlet = 1;
```

Routines for Creating Inlets

There are several functions for creating additional inlets, which are called in your object's creation function. You don't need to ever touch an inlet once its created, so these functions do not return pointers to the created inlets.

```
intin (void *myObj, int n)
```

`intin` creates integer inlets. The argument `myObj` should be a pointer to your object. The integer n, from 1 to 9, specifies the message type you'll get, so you can distinguish one inlet from another. For example, an integer sent in inlet 1 will be of message type in1 and a floating point number sent in inlet 4 will be of type ft4.

The order you create additional inlets is important. If you want the rightmost inlet to be the have the highest number in- or ft- message (which is usually the case), you should create the highest number message inlet first. For example, for four additional integer inlets (for a total of five):

```
intin(myobject,4);
intin(myobject,3);
intin(myobject,2);
intin(myobject,1);
```

```
floatin (void *x, int n)
```

Creates a floating point inlet. Analogous to `intin` for floating point numbers.

```
inlet_new (void *myObj, char *msg)
```

Creates a general purpose inlet. In general, you should try to make the "general purpose" inlet of your object the leftmost (automatically created) one, but you can use `inlet_new` for circumstances where you would like other messages than numbers to be received in additional inlets.

To specify a particular message that an inlet receives, pass the message's character string. For example, to create an inlet that receives only bangs, do the following...

```
inlet_new (myObject,"bang");
```

To create an inlet that can receive any message, pass `0L` for `msg`...

```
inlet_new (myObject,0L);
```

If you want an inlet to accept a list, you can write a method to respond to thelist message.

At initialization time:

```
addmess(myListMethod,"list",A_GIMME, 0);
```

Declare the method as:

```
void *myListMethod(x,s,ac,av)
struct myObject *x;
struct symbol *s;
int ac;
struct atom *av;
{

}
```

Routines for Creating Outlets

Your object is not created with an outlet automatically. Just like inlets, outlets can assigned types of integers, floating point numbers, or even specific messages. All the outlet creation functions listed below return a pointer to the created outlet and expect a pointer to your object as their first argument.

Since you need to reference outlet's explicitly, it makes sense to store a pointer to them inside your object. The leftmost outlet can be accessed as...

```
myObject->m_ob.o_outlet
```

...assuming `m_ob` is an Object which is the first field of your object.

`Outlet *bangout (void *x)`

Creates an outlet which will always send the bang message. You can send bang out a general purpose outlet, but using `bangout` allows Max to type check the connection a user might make. `bangout` returns the created outlet.

`Outlet *floatout (void *x)`

Creates a floating point outlet and returns the result.

```
Outlet *intout (void *x)
```

Creates an integer outlet for object x and returns the result. Example:

```
mynewobject->m_intout = intout(mynewobect);
/* create an outlet which will have integers sent through it */
```

```
Outlet *listout (void *x)
```

Creates a list outlet for object x and returns the result. See the discussion in the definition of `outlet_list` for information about lists.

```
Outlet *outlet_new (void *owner, char *type)
```

If you will be sending many data types and messages through an outlet, use the `outlet_new` function and pass `0L` for type. This creates the most basic type of outlet. If you want a specific type to be sent from an outlet, use `intout` (for integers), `bangout` (for bang messages), `listout` (for lists—see `outlet_list` for information about lists), or `floatout` (for floating point numbers).

Example:

```
void *outpointer;

outpointer = outlet_new (mynewobject,0L);    /* a general outlet */
```

Proxies

It's important to note that your object normally has no way of telling which of its inlets received a message, except for the translation which occurs between an int message and an in1, in2, in3 etc. message when you create an inlet with `intin`. (This also can occur with float messages using `floatin`).

However, there are some cases where you may want to be able to receive any message at all your inlets, and then be able to determine the inlet where message arrived. For example, consider a "multi-track recorder" object that wanted to use each inlet as an independent track. Rather than having to send messages to the leftmost inlet such as record 3 to put track 3 into record, you could send the record message directly into the third inlet. In addition, the recorder could "record" any kind of message that might arrive at each inlet, not just integers.

This noble dream can be accomplished when you use Proxy objects. Proxies are "super-inlets" that intercept messages arriving at an inlet before your object sees them. Then, after writing the inlet number where the message arrived into your object, the Proxy sends the message on to you, where you can check this inlet number and take appropriate action. You create a Proxy object with `proxy_new`, and unlike inlets and outlets, you must explicitly get rid of a Proxy (using `freeobject`) in your object's free function.

```
Proxy *proxy_new (struct object *owner, long id, long *stuffLoc)
```

Creates a new Proxy object (including an inlet). `owner` is a pointer to your object. `id` is the number you want written into your object when a message arrives in this proxy. It should be non-zero, and the convention is that it is the inlet "number" analagous to in1, in2 etc. Like inlets, your create

Proxies starting with the rightmost one. `stuffLoc` is the address of a location inside your object , which must be a long, where the Proxy's `id` number will be placed immediately before you receive the message from the Proxy. You should store the pointer returned by `proxy_new` because you'll need to free all Proxies in your object's free function.

After your method has finished, Proxy sets the `stuffLoc` location back to 0, since it never sees messages coming in an object's leftmost inlet. You'll know you received a message in the leftmost inlet if the contents of `stuffLoc` is 0.

Here's an example in which a Proxy is used to receive messages in three different inlets. Included is a sample bang method which prints out the inlet number where the bang message arrived.

Here's how we declare our object, making space for all our Proxy objects plus a long where the inlet number will be stored.

```
typedef struct {
    struct object m_ob;
    void m_proxy[2];        /* 3 inlets = 2 proxies */
    long m_inletNumber;     /* where proxy will put inlet number */
} myProxObject;
```

Here's the object creation function.

```
myProxObject_new()
{
    myProxObject *x;

    SetUpA4();
    x = (myProxObject *)newobject(class);

    /* create proxy objects from right to left */
    m_proxy[1] = proxy_new(x,2L,&x->m_inletNumber);
    m_proxy[0] = proxy_new(x,1L,&x->m_inletNumber);

    m_inletNumber = 0;      /* MUST clean this out at the beginning */
    RestoreA4();
    return (x);
}
```

Now here's the method written in response to a bang message.

```
void *myProxObject_bang(x)
myProxObject *x;
{
    SetUpA4();
    post("message arrived at inlet %ld",m_inletNumber);
    RestoreA4();
}
```

7. Writing Methods

This section documents four important structures you'll be using when writing your external object's methods. These are Outlets, Binbufs, Qelems, and Clocks. In addition, important utility routines you'll use to interface to Max are documented.

Routines for Using Outlets

These functions are used to send data out your object's outlets. You don't need to access the fields of an Outlet data structure, so use the `void *` type to refer to an outlet. Outlet is used here only for reasons of clarity.

```
void *outlet_anything (Outlet *x, Symbol *s, int argc, Atom *argv)
```

This is an advanced function which lets you send an arbitrary message out an outlet. `s` is the message you're sending (for integers, it would be the int symbol), and the arguments of the message are in argv. Most of the time, you won't be sending special messages from outlets, so you can use `outlet_int`, `outlet_list`, and `outlet_bang`. `outlet_anything` returns the result of sending the message (which you won't care about most of the time).

Here are a couple of examples of `outlet_anything`:

A hard way to send the bang message (see `outlet_bang` below for an easier way):

```
outlet_anything(myOutlet,gensym("bang"),0,NIL);
```

An even harder way to send a single integer:

```
Atom myNumber;

myNumber.a_type = A_LONG;              /* assign a type to the Atom */
myNumber.a_w.w_long = 432;             /* assign a value */

/* alternatively, you can use the macro SETLONG for the
    two lines above */

outlet_anything(myOutlet,gensym("int"),1,&myNumber);
```

Notice that `outlet_anything` expects the message argument as a *Symbol*, so you must use `gensym` (which transforms a C string into a Symbol) on a character string. If you'll be sending the same message a lot, you might call `gensym` on the message string at initialization time and store the result in a global variable to save the (significant) overhead of calling `gensym` every time you want to send a message. Also, it is incorrect to send lists using `outlet_anything` with list as the message Symbol. Use `outlet_list` instead.

```
void *outlet_bang (Outlet *x)
```

Sends a bang message out the outlet `x` and returns the result.

```
void *outlet_float (Outlet *x, float f)
```

Sends the floating point number `f` out the outlet `x` and returns the result.

```
void *outlet_int (Outlet *x, long n)
```

Sends the long integer `n` out the outlet `x` and returns the result.

```
void *outlet_list (Outlet *x, Symbol *s, int argc, Atom *argv)
```

Sends the list specified by `argv` and `argc` out the Outlet `x`. The Outlet must have been created with `listout` in your object creation function (see above). You create the list as an array of Atoms, but the first item in the list *must* be an integer or float. Existing Max objects (**pack**, **unpack**) can only deal with lists of integers and floating point numbers, so there isn't a great deal of use for lists of Symbols or mixed lists at the moment.

You may pass `gensym("list")` or 0L for the Symbol argument, which is ignored.

Here's an example of sending a list of three numbers.

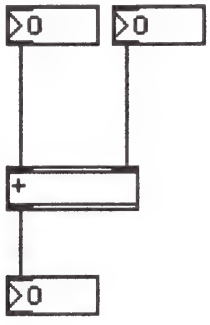
```
Atom myList[3];
long theNumbers[] = {32,35,77};
int i;

for (i=0; i < 3; i++) {
    SETLONG(myList+i,theNumbers[i]);  /* macro for setting an Atom */
}

outlet_list(myOutlet,0L,3,&myList);
```

It's not a good idea to pass large lists to `outlet_list` which are comprised of local (automatic) variables. If the list is small, as in the above example, there's no problem. If your object will regularly send lists, it might make sense to keep an array of Atoms inside your object itself.

Binbufs and Max Files

By choosing **Open As Text...** from the File menu, you can open a Max *binary* file as a text file. Fascinating, but what does it mean? Well, one thing you can use it for is changing Max binary files without opening them into Patcher windows. For another, it provides a window into the mechanism Max uses to save messages, called Binbufs (short for *bin*ary *buf*fer).

Binbufs are an "Atomized" counterpart of the Max text file. When you copy or duplicate part of a patch, it's turned into a binbuf. The Binbuf can then be "evaluated" because, lo and behold, it's just a bunch of Max messages. For example, here's a simple Max file that puts a + object and three number boxes into a patcher window, annotated line by line. (Note: Max is not guaranteed indefinitely into the future to produce exactly these lines when read in as text).




```
max v2;
```
- What version of Max file this is

```
#N vpatcher 50 38 450 338;
```
- Send a vpatcher message with window coords to the **new** object (abbreviated #N). The resulting new Patcher window is put on a "stack" so it can be referred to as #P symbol (internally, the `s_thing` field of the #P symbol is the newly created Patcher window object).

```
#P number 138 67 35 0;
```
- Send a number message to the Patcher, with arguments: coordinates of the number box.

```
#P number 98 67 35 0;
```
- Same as above

```
#P number 98 168 35 0;
```
- Same as above

```
#P newex 98 127 50 0 +;
```
- Send a newex message (the name for a box for making normal objects), with box coordinates. If you had typed any arguments after + they would appear after the +.

```
#P connect 0 0 1 0;
```
- connect messages tell the patcher to make connections between the previously defined objects. The numbers are backwards from the listed order (0 refers to the + object), so this line says, "Connect outlet 0 of object 0 to outlet 0 of object 1."

```
#P connect 3 0 0 1;
#P connect 2 0 0 0;
#P pop;
```
- Pop the current meaning of the #P symbol off the stack and restore the previous meaning (in this case, nothing).

When this file is read in, it's turned into a Binbuf, consisting of Atoms: symbols, numbers, semicolons, etc. This can then be evaluated as a message, where the first Symbol is the receiver, the second the message, and the additional Atoms are arguments to the message. A semicolon is used to separate messages.

Why would you want to know about Binbufs? One reason is to use them to evaluate text. Once a bunch of text has been transformed into a binbuf, it can be easily "evaluated" as a Max message. For example, the pop-up **umenu** object turns its text into a Binbuf, which can be evaluated as a message to be sent out its right outlet. The Binbuf can take care of separating a text stream into Atoms for you, generating Symbols and separating numbers from text as it goes. You'll also need to know about Binbufs if you want to do anything special to save the state of your object in a Max file. This is especially true if you're writing a user interface object (normal objects have their box coordinates and typed-in arguments saved for them).

You won't need to know about the internal structure of a Binbuf, so you can use the `void *` type to refer to one.

When writing a user interface object, you'll most likely use `binbuf_vinsert` to store your object's creation information. Additional details are furnished in the Writing User Interface Objects chapter.

Binbuf Routines

```
Binbuf *binbuf_new ()
```

Creates and initializes a Binbuf. Returns the created object if successful, 0 if not.

```
binbuf_append (Binbuf *x, Symbol *s, int argc, Atom *argv);
```

Append to the Binbuf `x` the symbol `s` followed by the array of Atoms pointed to by `argv`.

```
void *binbuf_eval (Binbuf *x, int argc, Atom *argv, void *receiver)
```

Evaluates the message in Binbuf `x` with arguments in `argv`, and sends it to `receiver`. Returns the result of sending the message.

```
binbuf_getatom (Binbuf *x, long *typeOffset, long *stuffOffset,
                Atom *result)
```

Fetches a single Atom from a Binbuf. `typeOffset` is the onset into the Binbuf's buffer of Atom types. `stuffOffset` is the onset into the Binbuf's binary data. To get the first Atom, set both `typeOffset` and `stuffOffset` to 0. These numbers are automatically incremented to point to the next Atom. `binbuf_getatom` returns 1 if there were no Atoms at the specified offsets, 0 if there's a legitimate Atom returned in `result`.

```
binbuf_insert (Binbuf *x, Symbol *s, int argc, Atom *argv)
```

`binbuf_insert` puts a Max message into a Binbuf, terminating it with a semicolon. You'd use this instead of `binbuf_append` if you were saving your object into a Binbuf and wanted a semicolon at the end. The Symbol will just be the first symbol in the message; hence it's the receiver of the message. `binbuf_vinsert` (see below) is probably a lot easier to use than `binbuf_insert`, since you don't have to format your data into an array of Atoms first.

`binbuf_insert` will convert the Symbols #1 through #9 into $1 through $9. This is used for saving patcher files which take arguments; it will probably never apply to anything you're doing.

```
void *binbuf_vinsert (Binbuf *x, char *fmtString, void types...)
```

`binbuf_vinsert` works somewhat like a `printf` for Binbufs. It allows you to pass a number of arguments of different types and insert them into a binbuf. The entire message will then be terminated with a semicolon. `fmtString` is a string containing the letters **s** (Symbol), **l** (long), or, **f** (float) in the order corresponding to the arguments you want placed in the Binbuf.

The example below shows a message to #N (the **new** object) of the symbol myobject followed by two numbers. In some way, this might be used to create a new instance of myobject. However, this is just a hypothetical use of binbuf_vinsert. For an actual example, see the section on the psave method in the Writing User Interface Objects chapter.

```
long arg1, arg2;
void *buf;

arg1 = 44;
arg2 = 20;
buf = binbuf_new();
binbuf_vinsert(buf,"ssll",gensym("#N"),gensym("myobject"),arg1,arg2);
```

```
void *binbuf_set (Binbuf *x, Symbol *s, int argc, Atom *argv)
```

Sets the Binbuf x to contain the message starting with s and continued by the Atoms in argv. Any previous Binbuf contents is destroyed.

```
void *binbuf_text (Binbuf *x, char **srcText, long textSize)
```

Parses the text in the handle srcText of length textSize into binbuf x. Use this to evaluate a text file or entry into a Binbuf.

Note: Commas, symbols containing a dollar sign followed by a number 1-9, and semicolons are identified by special *psuedo-type* constants for you when your text is binbuf-ized.

The following constants in the a_type field of Atoms returned by binbuf_getAtom identify the special symbols:

```
#define A_SEMI          10
#define A_COMMA         11
#define A_DOLLAR        12
```

For an Atom of the psuedo-type A_DOLLAR, the a_w.w_long field of the Atom contains the number the number contained after the dollar sign in the original text or symbol.

Using these psuedo-types may be helpful in separating "sentences" and "phrases" in the input language you design. For example, the pop-up **umenu** object allows users to have spaces in between words by requiring the menu items be separated by commas. It's reasonably easy to determine where each item starts by noting where the commas appear.

If you want to use a literal comma or semicolon in a symbol, precede it with a backslash (\) character. The backslash character can be included by using two backslashes in a row.

```
void *binbuf_totext (Binbuf *x, char **dstText, long *dstSize)
```

Converts the Binbuf x into text placed in the unlocked Handle dstText. Backslashes are added to protect literal commas and semicolons contained in symbols. The psuedo-types are converted into commas, semicolons, or dollar-sign and number, without preceding backslashes. binbuf_text can read the output of binbuf_totext and make the same Binbuf.

```
binbuf_read (Binbuf *x, char *filename, int volume, int binaryFlag)
```

Opens and reads the file referred to by `fileName` and `volume` into the already-existing Binbuf `x`. `binaryFlag` indicates whether the file is a Max-format binary file (file type 'maxb') or a Max-format text file (file type 'TEXT').

```
binbuf_write (Binbuf *x, char *filename, int volume, int binaryFlag)
```

Creates and writes the Binbuf `x` into a file called `fileName` on the specified volume. If `binaryFlag` is non-zero, the file will be saved as a Max-format binary file. If it's zero, the file is saved in a text format.

```
binbuf_free (Binbuf *x)
```

Frees the data associated with a Binbuf.

Clocks

Clock objects are your interface to Max's scheduler. There are two schedulers which run in parallel—the internal clock and the external clock. The internal clock runs in milliseconds. The external clock runs according to the current format in use in the MIDI Manager, which you can determine by calling `clock_getextfmt`.

To use the scheduler, you create a new Clock object using `clock_new` in your object creation function. You also have to write a Clock function, declared as follows:

```
myClockFun(x)
void *x;
```

The argument `x` is determined by the `arg` argument to `clock_new`. Almost always it will be pointer to your object.

Then, in one of your methods, use `clock_set` or `clock_delay` to schedule yourself on the internal clock, and `clock_xset` or `clock_xdelay` on the external clock. If you want unschedule yourself, call `clock_unset` (internal) or `clock_xunset` (external). To find out what time it is now (useful when calling clock_set), use `gettime` (internal), or `getxtime` (external).

The following routines are provided for Clock operations:

```
void *clock_new (void *arg, methodptr f)
```

Creates a new Clock object which will run function `f` with argument `arg` when it goes off. `arg` is usually a pointer to your object, which lets the function `f` access all the fields of your object's data structure.

```
clock_delay (Clock *x, long time)
```

Sets a Clock object to go off in `time` milliseconds from the current logical time using the internal clock.

`clock_xdelay (Clock *x, long time)`

Sets a Clock object to go off in `time` milliseconds from the current logical time using the external clock.

`int clock_getextfmt ()`

Returns the current external clock timing format. Values correspond to MIDI Manager clock format constants.

`clock_set (Clock *x, long time)`

Schedule Clock `x` for `time` (absolute) using the internal clock.

`clock_xset (Clock *x, long time)`

Schedule Clock `x` to go off at `time` (absolute) using the external clock.

`clock_unset (Clock *x)`

Take Clock `x` out of the internal scheduler.

`clock_xunset (Clock *x)`

Take Clock `x` out of the external scheduler.

`long gettime ()`

Returns the current logical time of the internal clock, which increments every millisecond.

`long getexttime ()`

Returns the current logical time of the external clock, in the current external clock format (see `clock_getextfmt`).

As an example, here's a fragment of how one might go about writing a metronome using the Max scheduler. First, here's the data structure we'll use.

```
typedef struct myMetro {
    void *m_clock;
    long m_interval;
    void *m_outlet;
} MyMetro;
```

We'll assume that the class has been initialized already. Here's the creation function that will allocate a new Clock.

```
void *myMetroCreate(defaultInterval)
long defaultInterval;
{
    MyMetro *x;
    void *myMetroClockFun();

    SetUpA4();
    x = (MyMetro *)newobject(class);      /* allocate space, etc. */
    x->m_clock = clock_new(x, myMetroClockFun);
        /* make a clock object */

    x->m_interval = defaultInterval;  /* store the interval */
    x->m_outlet = bangout(x);         /* outlet for ticks */
    RestoreA4();
    return (x);             /* return the new object */
}
```

Here's the method written to respond to the bang message.

```
myMetroBang(x)
MyMetro *x;
{
    SetUpA4();
    clock_set(x,gettime());
    RestoreA4();
}
```

Finally, here's the Clock function.

```
myMetroClockFun(x)
MyMetro *x;
{
    SetUpA4();
    clock_set(x,gettime()+x->m_interval);
                          /* schedule another metronome tick */

    outlet_bang(x->m_outlet);         /* send out a bang */
    RestoreA4();
}
```

```
clock_free (Clock *x)
```

Frees the space associated with a Clock object.

Qelems

Your object's methods may be called at interrupt level. This happens when the user chooses **Overdrive** from the Options menu and one of your methods is called from a scheduler Clock function. This means that you cannot count on doing certain things—like drawing, asking the user what file they would like opened, or calling any Macintosh toolbox trap which allocates or purges memory—from within an interrupt. The mechanism you'll use to get around this limitation is the *Qelem* (queue element) structure. Qelems also allow processor-intensive tasks to be done at a lower priority than in an interrupt. As an example, drawing on the screen, especially in color, takes a long time in comparison with many other tasks.

A Qelem works very much like a Clock. You create a new Qelem in your creation function with `qelem_new` and store a pointer to it in your object. Then you write a queue function, very much like the Clock function, which will get called from Max when the Qelem has been set. You set the Qelem to run its function by calling `qelem_set`.

Often you'll want to use Qelems and Clocks together. For example, suppose you want to update a counter on the screen 20 times a second. Write a Clock function that calls `qelem_set` and then reschedules itself for 50 milliseconds later (using the technique shown in the metronome example above). This scheme works even if you call `qelem_set` faster than the Macintosh can draw the counter, because if a Qelem is already set, it's not set again. When drawing the counter, you'll display its *current value*, not a specific value generated in the Clock function.

The following routines are provided for dealing with Qelems:

```
Qelem *qelem_new (void *arg, methodptr f)
```

Creates a new Qelem object which will run function `f` with argument `arg` at main level. Any kind of drawing or calling of Macintosh Toolbox routines that allocate or purge memory should be done in a Qelem function. `arg` is usually a pointer to your object, which lets the function `f` access all the fields of the object's data structure. You cause the qelem function to be run by calling `qelem_set`.

```
qelem_set (Qelem *q)
```

Causes the Qelem `q`'s function to be called with its argument at main level. If the Qelem object has already been set, it will not be set again. This is useful if you want to redraw the state of some data when it changes, but not in response to changes which occur faster than you can draw. A Qelem object is unset after its queue function has been called.

```
qelem_unset (Qelem *q)
```

If the Qelem `q`'s function is set to be called, `qelem_unset` will stop it from being called. Otherwise, `qelem_unset` does nothing.

```
qelem_front (Qelem *q)
```

Identical to `qelem_set`, except that the Qelem `q`'s function is placed at the front of the queue instead of the back. Be polite and only use `qelem_front` in special time-critical circumstances.

```
qelem_free (Qelem *q)
```

Frees a previously allocated Qelem object. You should call this in your free function if an instance of your class creates a Qelem.

Interrupt Level Considerations

Your object may be responding to messages at interrupt level, and there are a few guidelines you'll need to follow when writing methods so that your objects work correctly. Interrupt Level processing is enabled when the user chooses **Overdrive** from the Options menu. The advantages of Interrupt Level processing are the increased accuracy of timing, the ability for a Max patch to continue to operate smoothly while the user is using the menu bar or dragging a window, and the prioritization of faster timing and serial port operations over slower screen drawing. The basic rules for interrupt level operation are:

- Use the `lockout` mechanism when sending messages or calling outlet functions when *not* at Interrupt Level. This is normally only needed when writing a click method (needed in User Interface objects and objects that put up their own windows). If you use an outlet to send a message in a Qelem, you'll need to use `lockout` as well.

- Don't call Macintosh memory allocation routines directly in response to a message, or Macintosh traps which move or purge memory (see *Inside Macintosh*). Use the special Max routines `getbytes` and `freebytes`, and queue other types of memory management. Don't do anything which creates, adds to or frees a Binbuf or an Expr—these use Macintosh memory management calls.

- Use the Macintosh memory allocation calls `newhandle` and `disposhandle` instead of `_NewHandle` and `_DisposHandle`. These return an error and refuse to cooperate if a Memory Manager call was about to be made at interrupt level.

- Don't draw on the screen or put up a dialog box directly in response to a message. Use a Qelem. You can call `ouchstring` (for putting up error dialog box notices) in an interrupt, and the dialog will be queued to a lower priority for you.

Here are a few routines for dealing with Interrupt-driven processing issues:

```
int lockout_set (int newLockState)
```

Sets Max's global flag `lockout` to the value of `newLockState` and returns the previous setting of `lockout`. Use this to prevent any processing at interrupt level from occurring during a certain section of your code, typically calls to Outlet routines in response to click messages or inside Qelem functions. Here's a typical use of `lockout_set`:

```
int prevLock;

prevLock = lockout_set(1);
/* your critical region code here */
lockout_set(prevLock);
```

Always restore the lockout flag to its previous state by calling `lockout_set` again. If you don't, nothing will run at interrupt level ever again.

```
int isr()
```

Returns non-zero if you are at interrupt level, zero otherwise.

```
defer (Object *client, methodptr fun, Symbol *s, int argc, Atom
      *argv)
```

Calls the function `fun` with arguments `s`, `argc`, and `argv` immediately if you are not at interrupt level, otherwise, it will defer the call using `qelem_front`. You can use `defer` if you wish to respond to a message with arguments which may be sent to you at interrupt level. For example, the message box uses `defer` in its set method, which allows an incoming message to change the text of a message box even at interrupt level. Your function `fun` should be declared as follows:

```
void *myFun(client, s, argc, argv)
MyObject *client;
Symbol *s;
int argc;
Atom *argv;
```

```
defer_low (Object *client, methodptr fun, Symbol *s, int argc, Atom
                  *argv)
```

Always defers a call to the function `fun` whether you are at interrupt level or not, and uses `qelem_set`, not `qelem_front`. This is useful for responding to messages which will cause your object to open a dialog box, such as read and write.

Essential Max Utilities

Here are some important functions for interacting with the Max environment.

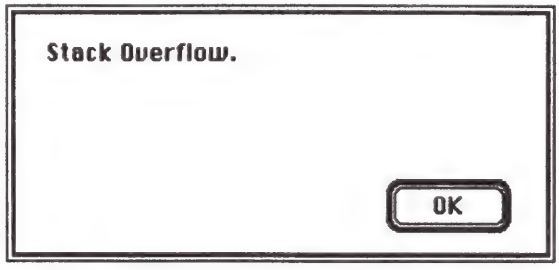
```
Symbol *gensym (char *string)
```

`gensym` takes a C string and returns a pointer to the Symbol associated with the string. Max maintains a symbol table of all strings to speed lookup for message passing. If you want to access the bang symbol for example, you'll have to use `gensym("bang")`. You may use `gensym` in writing a User Interface object's psave method, for saving extra data besides the object's box location and arguments. Or `gensym` may come into use in sending messages directly to other Max objects such as with `typedmess` and `object_anything`. These routines expect symbols—they won't `gensym` character strings for you.

```
ouchstring (char *fmtstring, void *items…)
```

Does an sprintf on `fmtstring` and `items`, then puts up an Alert box. `ouchstring` will queue the message to a lower priority level if it's called in an interrupt and there is no "ouch"-style message already pending.

An example of an ouch message you might have seen at one time or another:




```
post (char *fmtstring, void items...)
```

`post` is a printf-type function which writes a line in the Max window. It even works at interrupt level, queuing up to four lines to be printed at the main level. `post` can be quite useful in debugging your external object.

Note that `post` only passes 16 bytes of arguments to sprintf, so if you want additional items on a single line, use `postatom`.

Example:

```
int whatIsIt;

whatIsIt = 999;
post ("the variable is %ld",(long)whatIsIt);
```

Max Window output:

```
the variable is 999
```

```
error (char *fmtstring, void items…)
```

Prints sprintf-style into the Max window like `post`, but puts the attention-getting string

```
• error:
```

before the message.

Example:

```
error ("bad arguments to %s",myclassname);
```

Max Window output:

```
• error: bad arguments to myclass
```

```
postatom (Atom *a)
```

Prints a single Atom on a line in the Max window without performing a carriage return. The **print** object uses `postatom` to print lists.

```
freeobject (Object *x)
```

Calls the object x's free function, if any, then disposes the memory for x itself. You can use `freeobject` as a substitute for the various free functions listed throughout this document (i.e. `clock_free`, `qelem_free` etc.).

```
int sprintf (char *s, char *format, ...)
```

This is a special re-entrant version of sprintf which is provided so you can avoid linking sprintf into your external code resource. You can call it at interrupt level.

```
int sscanf (char *s, char *format, ...)
```

This is a special re-entrant version of sscanf which is provided so you can avoid linking sscanf into your external code resource. You can call it at interrupt level.

```
maxversion ()
```

Returns the version number of Max. Future versions of Max that add new external functions will have higher version numbers, so you'll be able to use this call to check if a specific routine is implemented. There's no reason to use `maxversion` to verify if any routine listed here is present.

```
assist_string (int rsrcID, long message, long arg, int firstin,
                int firstout, char *dstString, ...)
```

This routine is useful in your assist method. Given an existing `STR#` resource `rsrcID` (that you've copied to Max's temporary resource file with rescopy), `assist_string` sets `dstString` to a C string gathered from the resource at an index at `firstin` + `arg` (if message is `ASSIST_INLET`), or `firstout` + `arg` (if message is `ASSIST_OUTLET`). This allows you to store all your assistance strings in a single `STR#` resource. In addition, `assist_string` takes up to 16 bytes of arguments after `dstString` and will pass them to sprintf, so your STR# can display the state of variables in your object. Here's an example. Suppose we've stored the following two strings for our object in a STR# resource ID = 4534 (do not use resource indices between 4000 and 4100, 5000 and 5100 or 7000 and 7100 please—they are used by Max to display helpful text in various places). The strings are:

Sets the Value of the Bludgeon (Currently %ld)
Outputs a bludgeon Message

We can document this object in our assist method as follows:

```
myobject_assist(x,b,msg,arg,s)
MyObject *x;
void *b;
long msg,arg;
char *s;
{
    assist_string(4534,msg,arg,1,2,s,x->m_bludgeon);
}
```

Miscellaneous Operating System Utilities

Here are some functions provided for memory allocation and other low-level tasks.

```
int defvolume ()
```

Returns the vRefNum of the volume or folder in which the most recent file was opened.

```
disposhandle (Handle h)
```

Calls `_DisposHandle`. Returns a negative value if it is called at Interrupt Level (in which case it refuses to dispose the Handle).

```
drawstr (char *s)
```

Draws a C string at the current pen location using the current font and size.

```
freebytes (void *p, int size)
```

Frees memory allocated with `getbytes` and places it in Max's free list. Works at interrupt level. Notice that you must pass the size of the pointer to be freed in bytes.

```
void *getbytes (int size)
```

A substitute for `_NewPtr` which takes the memory from Max's own free memory list and does error checking. 0 is returned if the memory isn't available. size is specified in bytes. `getbytes` works at interrupt level.

```
growhandle (Handle h, long size)
```

A substitute for `_SetHandleSize` with some error checking which refuses to work at interrupt level.

```
char **newhandle (long size)
```

Like `_NewHandle`, but won't call `_NewHandle` if you're at interrupt level and does some error checking.

Routines for Dealing with Files

```
open_dialog (char *filename, int *vol, OSType *dstType,
            SFTypeList typelist, int numtypes)
```

This is convenient wrapper around the Standard File Dialog for opening files. You pass a standard list of file types in `typelist` as you would to `_SFGetFile`, and a number of types (up to 4) in `numtypes`. `open_dialog` returns 0 if the user clicked Open in the dialog box, and returns the name of the file picked in `filename`, its vRefNum in `vol`, and its file type in `dstType`.

The standard types for Max files are 'maxb' for binary files and 'TEXT' for text files.

```
saveas_dialog (char *filename, int *vol, int *binary)
```

This is a convenient wrapper around the Standard File Dialog for saving files. It's most useful when saving Binbufs, since it provides the user with the option of saving the Binbuf as text or in a binary format. Pass a default name for saving the file in `filename`. `saveas_dialog` returns 0 if the user clicked Save in the dialog box, non-zero otherwise. The name to use to save the file is returned in `filename`, along with the disk or directory's vRefNum in `vol`. `binary` will be 1 if the user wishes to save the file in binary format, 0 if the file is to be saved as text. If you pass `0L` for `binary`, the option to save the file as text will not be presented (you can of course save the file with any type and creator you wish).

```
locatefile (char *filename, int *vol, int *binary)
```

Searches through the directories specified by the user for Patcher files and tables the File Preferences dialog as well as the "default volume" (see `defvolume`) and the directory containing the Max application. If a file is found with the name `filename`, `locatefile` returns 0, otherwise it returns non-zero. The file's vRefNum (volume reference number) is returned in `vol`, and `binary` is non-zero if file is in Max binary format, 0 if it's in text format. `filename` and `vol` can then be passed to `binbuf_read` to read and open file the file. Note: the arguments to `locatefile` have changed since the last revision of this document—a second character string argument has been removed.

```
locatefiletype (char *filename, int *vol, OSType filetype,
            OSType creator)
```

Searches through the same directories as `locatefile`, but allows you to specify a type and creator of your own. `locatefile`, in contrast, searches for only the standard types of Max files. If you pass `0L` for `filetype` or `creator`, `locatefiletype` doesn't try to match the filetype and/or creator of any file it finds named `filename`.

```
int serialno ()
```

Returns a serial number which is incremented each time a Patcher file is saved. This routine is useful for objects like **table** and **coll** which have multiple objects that refer to the same data, and can "embed" the data inside a Patcher file. If the serial number hasn't changed since your object was last saved, you can avoid saving multiple copies of the object's data.

8. Advanced Facilities

The following sections discuss capabilities that most objects will not need to use, but may be of interest to advanced programmers.

Advanced Object Creation and Message Routines

```
Object *newinstance (Symbol *className, int argc, Atom *argv)
```

Creates a new instance of the class specified by the Symbol `className` with additional arguments in `argv`. Using `newinstance` is equivalent to typing something in a New Object box when using Max. The difference is that no object box is created in any Patcher window, and you can send messages to the object directly without connecting any patch cords. The messages can either be type-checked (using `typedmess`) or non-type-checked (using the members of the `getfn` family). `newinstance` returns a pointer to the created object. This function is useful for taking advantage of other objects which may have been defined externally which you would like to use "privately" in your object, such as tables. Documentation about how to actually go about doing this is forthcoming, but for now, you can at least do cute things such the useless example shown under `zgetfn`.

`newinstance` returns a pointer to the created object, or 0 if there was an error.

```
void *typedmess (void *receiver, Symbol *message, int argc,
      Atom *argv)
```

`typedmess` is an advanced function which sends a Max object (`receiver`) a message with arguments. If the `receiver` object can respond to the message, `typedmess` returns the result. Otherwise, an error message will be seen in the Max window and 0 will be returned. Note that the message must be a Symbol, not a character string, so you must call `gensym` on a string before passing it to `typedmess`.

Example:

If I want to send a bang message to the object `bang_me`...

```
void *bangResult;

bangResult = typedmess(bang_me,gensym("bang"),0,0L);
```

The following three routines send messages to objects which are not type-checked. You are responsible for passing the message arguments correctly. System errors, not error messages in the Max Window, will result if you don't. Typically, these functions will be useful with an object created with `newinstance`.

```
fptr getfn (Object *x, Symbol *msg)
```

Returns a pointer to the method bound to the message `msg` in Object `x`'s message list. Returns 0 and prints an error message in Max Window is the specified method doesn't exist.

```
fptr egetfn (Object *x, Symbol *msg)
```

Returns a pointer to the method bound to the message `msg` in Object `x`'s message list. Returns a pointer to a "null function" (one which exists but does nothing) if the specified method doesn't exist.

```
fptr zgetfn (Object *x, Symbol *msg)
```

Returns a pointer to the method bound to the message `msg` in Object `x`'s message list. Returns 0 if the specified method doesn't exist, but doesn't print an error message.

The macros `mess0`, `mess1`, `mess2`, etc. defined in **ext_mess.h** are useful "wrappers" for sending non type-checked messages. Here's a silly but illustrative example of using `newinstance` and non type-checked messages.

In this example, the class **bob**'s info method creates an instance of the **joe** class, sends it an info message (which presumably does something) and destroys the instance of **joe**.

```
void *bob_info(x,p,b)
Bob *x;
void *p;
void *b;
{
    void *joeboy;

    SetUpA4();

    joeboy = newinstance(gensym("joe"),0,0L);      /* create a joe */
    mess2(joeboy,gensym("info"),p,b);     /* send untyped message */
    freeobject(joeboy);                   /* kiss joe goodbye */

    RestoreA4();
}
```

Table Access

You can use these functions to access named **table** objects. Tables have names when the user creates a **table** with an argument, such as...

The scenario for knowing the name of a **table** but not the object itself is if you were passed a Symbol, either as an argument to your creation function or in some message, with the implication being "do your thing with the data in the **table** named norris."

```
table_get (Symbol *tableName, long ***dstHandle, long *dstSize)
```

`table_get` searches for a **table** associated with the Symbol `tableName`. If one is found, it returns a Handle to its elements (stored as an array of long integers) in `dstHandle` and the **table**'s size in `dstSize`. If no table is associated with the symbol `tableName`, `table_get` returns a non-zero result.

```
table_dirty (Symbol *tableName)
```

Given the name of a **table** in `tableName`, `table_dirty` sets its dirty bit, so the user will asked to save changes if the **table** is closed. If no **table** is associated with `tableName`, `table_dirty` returns a non-zero result.

Text Editor Windows

Max has a simple built-in text editor object *Ed*, which can display and edit text from your object. The routines described below let you create a text editor.

When the editor window is about to be closed, you will potentially receive two messages. The first one, okclose, will be sent if the user has changed the text in the window. This is the standard okclose message that is sent to all "dirty" windows when they are about to be closed, but the text editor window object passes it on to you instead of doing anything itself. Refer to the section on Window Messages for a description of how to write a method for the okclose message. It's not required that you write one—if you don't, the behavior of the window will be determined by the setting of the window's `w_scratch` bit (described in the Doing Windows chapter). If it's set, no confirmation will be asked when a dirty window is closed (and no okclose message will be sent to the text editor either).

The second message, edclose, requires a method, which should be added to your object at initialization time:

```
addmess(myobject_edclose,"edclose", A_CANT, 0);
```

edclose will be sent to your object in the following two cases:

- The window's `w_scratch` bit has been set
- The user clicked on **Yes** in the Alert placed on the screen when the window was about to be closed. The window's `w_scratch` bit was not set in this case.

Define your edclose method as follows:

```
void *myobject_edclose (x,textHandle,textSize)
Myobject *x;
char **textHandle;
long textSize;
{

}
```

In this method, the editor will hand you the text in the editing window in `textHandle` and its size (in characters) in `textSize`. If you set the window's `w_scratch` bit, you might want to know if the text in the window was actually modified by the user. The definition of the text editing object Ed is in **ext_edit.h**. Use the following code to check the dirty bit of the editor's window (assume `x->m_edit` points to your text editor object):

```
if (x->m_edit->e_wind->w_dirty)
    /* has been modified */
```

In the case where the text has been modified, you can update the state of your object using the text arguments of the edclose message.

After receiving an edclose message, the text editor window is destroyed, so it's important to note this in the internal state of your object. It's a good idea to set the field of your object that points to the text editor to 0 at this point.

```
Ed *ed_new (Object *assoc)
```

Creates a new text editing window with associated object `assoc`. The text editor window will be immediately visible. `assoc` should be a pointer to your object, so it can receive the all-important edclose message. You should store the result of `ed_new` in your object. Two reasons: so you can call `ed_settext` to set the text in the window and `ed_vis` to bring the window to the front when your object receives a dblclick message.

```
ed_settext (Ed *x, Handle textHandle, long textSize)
```

Set the text in the text editing window x to the first textSize characters in textHandle. The window will be refreshed to show the new text.

```
ed_vis (Ed *x)
```

Call `ed_vis` if you want to bring an existing text editor window to the front. If you want to implement the standard behavior for text editor windows, you'll make yours visible only after receiving a dblclick message. Since the editor window is immediately visible when created with `ed_new`, don't create an editor in your object creation function. Rather, initialize the slot for storing your editor to 0 when your object is created. Then you can determine whether you need to create a new editor in your dblclick method, or merely bring one that already exists to the front. Here's an example dblclick method. We assume that `x->m_edit` is a field in your object which contains a text editor.

```
myobject_dblclick(x)
Myobject *x;
{
    SetUpA4();

    if (x->m_edit)
        ed_vis(x->m_edit);
    else {
        x->m_edit = ed_new(x);
        /* set text of editor here with ed_settext */
    }

    RestoreA4();
}
```

```
ed_free (Ed *x)
```

Disposes of the memory in a text editor object.

Access to **expr** Objects

If you want to use C-like variable expressions which are entered by a user of your object, you can use the "guts" of Max's **expr** object in your object. For example, the **if** object uses **expr** routines for evaluating a conditional expression, so it can decide whether to send the message after the words then or else. The following functions provide an interface to **expr**.

Note: As with Binbuf, Clock, and Qelem, You don't need to declare variables of type Expr, just use pointer to void. The types are shown here for reference only. Constants and other declarations needed to use Expr are found in **ext_expr.h**.

```
Expr *expr_new (int argc, Atom *argv, Atom *result)
```

Creates an Expr object from the arguments in `argv` and returns the type of any Expr-style arguments contained in `argv` (i.e. $i1, etc.) in atoms in an array pointed to by `result`. `result` should already exist as an array of 9 Atoms, all of which are filled in. If an argument was not present, it will set to type `A_NOTHING`. For example, suppose `argv` pointed to the following atoms:

```
$i1     (A_SYM)
+       (A_SYM)
$f3     (A_SYM)
+       (A_SYM)
3       (A_LONG)
```

After calling `expr_new`, `result` would contain the following:

Array Index	*Argument Number*	*Type*	*Value*
0	1 ($i1)	A_LONG	0
1	2	A_NOTHING	0
2	3 ($f3)	A_FLOAT	0.0
3	4	A_NOTHING	0
4	5	A_NOTHING	0
5	6	A_NOTHING	0
6	7	A_NOTHING	0
7	8	A_NOTHING	0
8	9	A_NOTHING	0

```
void *expr_eval (Expr *x, int argc, Atom *argv, Atom *result)
```

Evaluates the expression in `x` with arguments in `argv` and returns the type and value of the evaluated expression as an Atom in `result`. `result` need only point to a single Atom, but `argv` should contain at least `argc` Atoms. If, as in the example shown above under `expr_new`, there are "gaps" between arguments, they should be filled in with Atoms of type `A_NOTHING`.

```
expr_free (Expr *x)
```

Frees the Expr object pointed to by `x`.

Presets

Max contains a special **preset** object, which has the ability to send preset messages to some or all of the objects (*clients*) in a Patcher window. The preset message, sent when the user is *storing* a preset, is just a request for your object to tell the preset object how to restore your internal state to what it is now. Later, when the user *executes* a preset, the **preset** object will send you back the message you had previously specified.

The dialog goes something like this:

• During a store...

preset *object object to Client object(s)*: hello, this is the preset message—tell me how to restore your state

Client object to **preset** *object:* send me int 34 (for example)

• During an execute...

preset *object to Client object*: int 34

The client object won't know the difference between receiving int 34 from a **preset** object and receiving a 34 in its leftmost inlet.

It's not mandatory for your object to respond to the preset message, but it is something that will make users happy. All Max user interface objects currently respond to preset messages.

Here are routines you can use when responding to the preset message:

```
preset_store (char *format, void items...)
```

This is a general preset function for use when your object's state cannot be restored with a simple int or set message. The arguments are essentially the same as those you would pass to `binbuf_vinsert` (see above). The example below shows the expected format for specifying what your current state is to a **preset** object. The first thing you supply is your object itself, followed by the symbol which is the name of your object's class (which you can retrieve from your object using the macro `ob_sym`, declared in `ext_mess.h`). Next, supply the symbol which specifies the message you want receive (a method for which had better be defined in your class), followed by the arguments to this message—the current values of your object's fields.

Here's an example of using `preset_store` which specifies that the object would like to receive a set message. We assume it has one field, `myvalue`, that it would like to save and restore.

```
void *myobject_preset(x)
struct myobject *x;
{
    SetUpA4();

    preset_store("ossl",x,ob_sym(x),gensym("set"),x->myvalue);

    RestoreA4();
}
```

When this preset is executed, you'll receive a set message whose argument will be the value of `myvalue`.

Note that the same thing can be accomplished more easily with `preset_set` and `preset_int`.

Don't pass thousands of items to `preset_store`. If you want to store a huge amount of data in a preset, use `binbuf_insert`. The following example locates the Binbuf into which the preset data is being collected, then calls `binbuf_insert` on a previously prepared array of Atoms. We'll assume that the state of your object can be restored with a set message.

```
myobject_preset(x)
myobject *x;
{
    void *preset_buf;       /* Binbuf that stores the preset */
    int atomCount;          /* number of atoms you're storing */
    Atom atomArray[SOMESIZE];     /* array of atoms to be stored */

    SetUpA4();

    /* 1. prepare the preset "header" information */
    SETOBJ(atomArray,x);
    SETSYM(atomArray+1,ob_sym(x));
    SETSYM(atomArray+2,gensym("set"));

/* >> fill atomArray with object's state here and set atomCount << */

    /* 2. find the Binbuf */
    preset_buf = gensym("_preset")->s_thing;

    /* 3. store the data */
    if (preset_buf) {
        binbuf_insert(preset_buf,NIL,atomCount,atomArray);
    }

    RestoreA4();
}
```

```
preset_set (Object *x, long value)
```

An easier way to respond to the preset message if your object would like the set message with a single integer to restore its state.

```
preset_int (Object *x, long value)
```

This function is the same as `preset_set` but sends the int message instead (see the Messages chapter for the difference between int and set). Four out of five doctors prefer the int message to the set message for restoring an object's state when executing a preset.

Event Serial Numbers

If you call `outlet_int`, `outlet_float`, `outlet_list`, or `outlet_anything` inside a Qelem or during some idle or interrupt time, you should increment Max's Event Serial Number beforehand. This number can be read by objects that want to know if two messages they have received occurred at the same logical "time" (in response to the same event). Max increments the serial number for each tick of the clock, each key press, mouse click, and MIDI event.

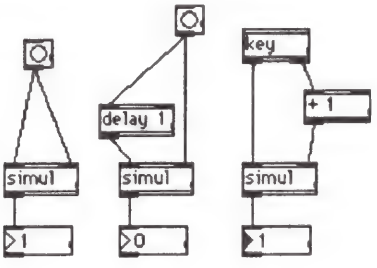
```
evnum_incr ()
```

Increments Max's Event Serial Number.

```
long evnum_get ()
```

Returns the current value of the Event Serial Number.

Here is a Max patch which includes an object called **simul** which, using `evnum_get`, returns a 1 if the right and left inlets receive messages at the same time, 0 if not. The number boxes below show the results of clicking on the **button** objects or typing a key.



MIDI Manager Access

```
midiinfo (MidiInfoRec *midiWorld)
```

You can make calls to the MIDI Manager directly from an external object using the Glue Routines in **MIDIGlueA4.π**. This is a special version of the MIDI Manager glue intended for use with A4 Projects such as your external object code resource. Before making any MIDI Manager calls, call `midiinfo` to find out whether Max is in fact signed into the MIDI Manager, as well as the refNums of all Max's MIDI Manager data and time ports. `midiinfo` returns its information into a `MidiInfoRec` data structure declared as follows:

```
typedef struct {
    int usingCalamari;
    int nInPorts;
    int nOutPorts;
    int inRefNum[16];
    int outRefNum[16];
    int intTimeRefNum;
    int timeInRefNum;
    int timeOutRefNum;
    OSType applSig;
} MidiInfoRec;
```

If `usingCalamari` is 0 (meaning Max is not using the MIDI Manager), the rest of the data is undefined. If it's non-zero, Max is using the MIDI Manager and the rest of the `MidiInfoRec` is valid. `nInPorts` and `nOutPorts` are the total number of input and output data ports in use. `inRefNum` and `outRefNum` contain the refNums of the input and output data ports. Note that there is space for 16 such ports for input and output, but there will only be `nInPorts` or `nOutPorts` valid refNums. `intTimeRefNum` is the internal millisecond time port used by Max. `timeInRefNum`

is the input time port used by the "external clock" scheduler (see the section on Clocks above). `timeOutRefNum`, the output time port, is used by the external Max object **timeout**. `applSig` is the signature this copy of Max used to sign into the MIDI Manager. It might not be 'max2' if more than one copy of Max is running simultaneously.

Before calling MIDI Manager routines, you need to call `GetMIDIMgrTrapAddress`, a routine contained in MIDIGlueA4.π. `GetMIDIMgrTrapAddress` takes no arguments. In addition, make sure you've called `_SetupA4` before accessing any routines in MIDIGlueA4.π.

9. Doing Windows

Max allows external objects to create their own windows and handle Macintosh events. Generally, these tools simplify the task of writing a user interface, and they're a bit simpler to manage than writing user interface objects that exist within Patcher windows.

If your window will be an "editor" for a Max object, you should open it when your object receives a dblclick message (Max doesn't send normal objects a single click message).

Your object will be able to respond to window messages because Max will install a reference to it inside each Macintosh window record. When Max detects a Macintosh event in the window, it sends the appropriate message to the object that "owns" the window.

There are two things you'll need to know in order to work with windows in Max.

- A set of Max functions you'll use to allow your window to exist within the Max world.
- A set of special window messages your object will be sent while its window is open. You write methods to respond to these messages perform actions like drawing the contents of the window or handling a mouse click. At initialization time, use `addmess` to install these methods in your class. All of the window methods should use the special argument list...

  ```
  A_CANT, 0
  ```

  ...which specifies that Max can't and shouldn't typecheck the arguments of the message (because nothing is being passed as a type-checkable Atom).

Note that none of the window messages will ever be sent to you at interrupt level.

The basic window structure is called a Windpub (short for *pub*lic *wind*ow—the Wind type contains additional "private" fields). You'll create one of these in your creation function (or whenever you want a window to open) by calling `wind_new`. Then all you have to do is sit back and wait for messages. The Windpub and the flags to pass to `wind_new` are declared in the include file **ext_wind.h**.

The messages sent by the window system are listed on the following pages. In each case, you should frame any action inside calls to set the current GrafPort to your window using `wind_setport`. The example below assumes a pointer to the Windpub returned from calling `wind_new` is stored in the objects `m_wind` field.

```
GrafPtr sp;

SetUpA4();
if (sp = wind_setport(myObject->m_wind)) {

    /* respond to the window message here */

    SetPort(sp);
}
RestoreA4();
```

Note that you can't call `_SetPort` directly on the Windpub pointer—it doesn't point to a `GrafPort`.

Basic Window Messages

The function definition shows how your method to respond to the listed message should be declared.

The click message:

```
myObject_click (MyObject *x, Point pt, int dblClickFlag, int modifiers)
```

`pt` is the location of the mouse click in coordinates local to the window. `dblClickFlag` is non-zero if Max has detected a double-click. `modifiers` is the standard Macintosh modifiers field of an `EventRecord` for detecting the presence of the option, command, shift, caps lock, or control keys.

Your click method should handle a mouse event at the specified location. See the Max functions `wind_defaultscroll` and `wind_drag` below for some assistance in this task.

The update message:

```
myObject_update (MyObject *x)
```

Respond to an update event by drawing the contents of the window. Note that Max takes care of drawing the scroll bars (Max will call `_DrawControls` on your window) and grow icon of your window, if it contains those items. Also, the affected area of the window will have been erased for you.

The key message:

```
myObject_key (MyObject *x, int k, int modifiers, int keycode)
```

Respond to key down or auto key event. The ASCII code of the key is passed as argument `k`, along with the standard Macintosh modifiers field of an EventRecord, and the Macintosh key code.

The idle message:

```
myObject_idle (MyObject *x, Point mouseLoc, int within)
```

Your idle method is called repeatedly while your window is the active window, or, when the user has chosen **All Windows Active** from the Options menu and the cursor happens to be over your window. The location of the mouse is passed in local coordinates in `mouseLoc`, so you won't have to call `_GetMouse` to find it. Typically, windows will use the idle method to adjust the cursor when it falls over specific locations, such as TextEdit fields. However, do not change the appearance of the cursor unless `within` is non-zero. `within` will be zero if your window is active but the cursor is not over it. You can set the cursor with `wind_setcursor`.

The patcher window also uses its idle method to call `_TEIdle` if text is being edited, and to highlight inlets and outlets if the mouse is over them. Note that your idle function is also called during the time the mouse is down and you've called `wind_drag`.

The activate message:

```
myObject_activate (MyObject *x, int activeFlag)
```

Respond to an activate event. `activeFlag` is non-zero if this is an activate event, or zero if it's a deactivate event.

The close message:

```
myObject_close (MyObject *x)
```

This message is sent to your object when the user has clicked on the close box of its window. It's suggested that you write your object so that closing the window will not destroy your object's data, so the user can close the window without worrying about losing anything. If you plan on creating another window with `wind_new` at a later time, dispose of the window's memory by calling `wind_free` (which will call the `_CloseWindow` trap). Otherwise, you might just call `_HideWindow`. Max just tells you that the user wants to close your window. You can respond in any manner you like.

Advanced Window Messages

The scroll message:

```
myObject_scroll (MyObject *x)
```

Max calls this routine when the user is moving a scroll bar. You should check the `w_xoffset` (horizontal) and `w_yoffset` (vertical) fields of your `Windpub`, compare against your stored previous values, and scroll the window accordingly.

The vis and invis messages:

```
myObject_vis (MyObject *x)
myObject_invis (MyObject *x)
```

Max sends you these messages when you window has just become visible or is just about to become invisible. You can perform any appropriate action at this time, such as initializing the window's user interface objects when receiving the vis message.

The oksize message:

```
myObject_oksize (MyObject *x, int *hSize, int *vSize)
```

If you wish, you can implement an oksize method for your window, which will allow you to check and possibly adjust the size of a window before it is actually resized (see the wsize message below). The proposed size is passed in `hSize` and `vSize`. You can set these to whatever value you like. You might use the oksize message if your window contains "cells" and you want the size of the window to be an exact multiple of the number of cells.

The wsize message:

```
myObject_wsize (MyObject *x, int hSize, int vSize)
```

If the user resizes a window, you'll be informed via the wsize message. The new dimensions of the window are passed in `hSize` and `vSize`. If your window has scroll bars, you will need to resize, recalibrate, and redraw them here.

The otclick message:

```
myObject_otclick (MyObject *x, Point globalPoint)
```

If you wish, you can respond to the otclick message, which will be sent to your window if the user option-clicks on the title bar of a window (otclick stands for Option-Title-Click). As an example, the Patcher window implements a pop-up menu that allows you to return to the parent window of a subpatch. If you don't implement an otclick method, the user will get the normal dragging action that you'd expect when clicking on a title bar.

Menu Messages

The chkmenu message:

```
myObject_chkmenu (MyObject *x, Menuinfo *mi)
```

When your window becomes the active window, Max will update its menus based on those items your window could respond to. You are passed a pointer to a `Menuinfo` structure initialized to all zero values. If you wish to be sent a message when the user chooses a particular menu item, set the corresponding item in the `Menuinfo` structure to 1, and the item will be enabled. The `Menuinfo` structure is an array of integers. Below are the menu commands available to you:

Offset	*Message*	*Commands*	*Additional Notes*
0	cut, copy, clr	Cut, Copy, and Clear – Edit menu	
2	paste	Paste – Edit menu	
4	dup	Duplicate – Edit menu	
6	saveto	Save and Save As… – File menu	See below
8	pastepic	Paste Picture – Edit menu	See below
10	edit	Edit… – Max menu	
12	dialog	Get Info… – Max menu	
14	lineup	Align – Max menu	
16	fixwidth	Fix Width – Max menu	
18	font	The entire Font menu	See below
20	hide	Hide on Lock – Max menu	
22	show	Show on Lock – Max menu	
24	selectall	Select All – Edit menu	
26	find	Find… – Edit menu	See below
28	find	Find Again – Edit menu	See below
30	find	Replace – Edit menu	See below
32	print	Print… – File menu	See below

All of these commands send messages with your object as the first argument. Only the saveto and find messages contain additional arguments, detailed below.

The undo message:

Called when the user chooses **Undo** from the Edit menu. You need to explicitly enable and set the text of the Undo menu item, which you can do with `wind_setundo` discussed below. It's your responsibility to keep track of whether this message represents an undo or a redo, but this shouldn't be hard, since you set the text of the menu item yourself. If you never call `wind_setundo`, the Undo menu item will always be disabled when your window is the active window.

The pastepic message:

Called when the user chooses **Paste Picture** from the Edit menu. The picture data is still on the clipboard, and it's your responsibility to copy the data (using `_GetScrap`) and do something with it.

The saveto message:

```
myObject_saveto (MyObject *x, char *fileName, int vRefNum)
```

If the user chooses **Save As...** from the File menu when your window is the active window, you will receive this message after the user has specified a file name and volume in a standard save file dialog box. The file has been neither opened nor created. You should create, open, write out, and close the file. You might also want to change the title of the window using `wind_settitle`, although this is done automatically unless you set the `WKEEPT` bit in the flags argument passed to `wind_new`.

If the user chooses **Save** from the File menu, the filename and volume are already known to Max, and your saveto method will be called.

You're not required to implement this method if there is nothing worth saving in your window. In this case, don't enable the saveto message in your chkmenu method.

The dialog message:

The dialog method is used in the Patcher window to send the info message to any selected Patcher objects. Your object can respond to the info message, as we've mentioned above, usually by putting up a dialog box to set some internal values.

The dialog message will be sent to you when your window is the active window and the user chooses **Get Info...** from the Max menu. Your method might put up the same dialog box as your info method.

The font message:

Your object will receive this message after the user has changed the window's default font or font size by choosing from the Size menu. You can determine the current size by checking the `w_fontsize` field of your Windpub. Currently, there is no record of the default font, but there might be soon.

The help message:

The help message is sent when the user chooses **Help...** from the Max menu when your object's window is the active window. Your help method can respond by opening a help file or any other helpful action.

The find message:

This message will be sent to your window after the user has clicked **Find** in the Find Dialog with your window active, or chosen **Find Again** or **Replace** from the Edit menu. Your method should be declared as follows:

```
myObject_find (MyObject *x, void *search, void *replace)
```

If you actually "find" something, you should probably enable the **Find Again...** item the next time you receive a chkmenu message.

The arguments `search` and `replace` are Binbufs. If `replace` is 0L, you are only to search. If `search` is 0L, you are only to replace. If both `search` and `replace` are not 0L, first replace what's selected in your window, then search again. If you want to change `search` or `replace` to text, use `binbuf_totext`.

The print message:

```
myObject_print (MyObject *x, THPrint hPrint, GrafPtr yourPort,
            int *result)
```

This message is sent to your window when the user has chosen **Print...** from the File menu, seen the standard printing dialog and clicked OK. You will receive a print message for every copy of the document the user wants to print.

In this method, you print your window as through it were a "document." `hPrint` is a standard Macintosh THPrint record as described in *Inside Macintosh*, Vol II, page 149. `yourPort` is your window's GrafPort (you won't print in this port, but it may be convenient to know). `result` is a pointer to an integer in which you can return the status of the printing. If there's an error, set `result` to a non-zero value. It's set to zero before your method is called, so you can leave it alone if all goes well.

Here is an outline of some simple code that would print out a Patcher window on a single page (too bad if the window is too big). Note the specific calls to the Printing Manager you are required to make before and after drawing a page.

```
void *patcher_print(x,hPrint,myPort,result)
struct patcher *x;
THPrint hPrint;
GrafPtr myPort;
int *result;
{
    TPPrPort printPort;
    Rect printRect;

    printPort = PrOpenDoc(hPrint,0L,0L);
    SetPort(printPort);
    TextFont(myPort->txFont);
    TextSize(myPort->txSize);
    printRect = (**hPrint).prInfo.rPage;
    PrOpenPage(printPort,0L);

    /* print patcher here -- just draw all the boxes and lines */

    PrClosePage(printPort);
    PrCloseDoc(printPort);
    *result = 0;
}
```

You may want to be nice and add code that checks for command-period being typed. In this case, `result` should be set to a non-zero value, so that additional copies of the document are not printed.

Window Routines

```
Windpub *wind_new (void *assocObj, int left, int top, int right,
      int bottom, int flags)
```

Creates a new Max window. `left`, `top`, `right`, and `bottom` are in global coordinates. `assocObj` should be a pointer to your object. This is the object which will receive all the window's messages. `flags` is a combination of the following constants:

```
#define WVIS            1       /* Window will be visible */
#define WGROW           2       /* has a Grow region */
#define WSCROLLX        4       /* horizontal scrollbar */
#define WSCROLLY        8       /* vertical scrollbar */
#define WCLOSE          16      /* has a close box */
#define WKEEPT          32      /* keep window title from Save As */
#define WSGROW          64      /* invisible grow region */
#define WCOLOR          128     /* Color window (if Mac II) */
#define WPATCHPROC      256     /* patcher wdef w/ extra title gadgets */
#define WSHADOWPROC     512     /* procID used in new obj list window */
```

The actual Macintosh window will not be created or appear on the screen unless the visible flag is set.

```
wind_vis (struct windpub *x)
```

Makes the window visible. If it's already visible, `wind_vis` calls `_SelectWindow` to make the window the active window.

```
wind_invis (Windpub *x)
```

Hides the window if it's visible. If yours is the only object to reference the window, the window is permanently closed. If the window isn't visible, `wind_invis` does nothing. If you want to defeat the system and keep your window alive but invisible, call `_HideWindow` instead. Note that `wind_invis` does not actually get rid of the memory occupied by the Windpub structure. Call `wind_vis` to create another Macintosh window.

```
int wind_defaultscroll (Windpub *x, Point pt, int pageSize)
```

If your window has scroll bars, call `wind_defaultscroll` in your click method. It will check if `pt` lies within a scroll bar. If so, `wind_defaultscroll` executes the default scroll bar routine and returns 1. If not, `wind_defaultscroll` returns 0. If you pass 0 for `pageSize`, the default paging routine is used, which essentially goes to the top or bottom of the scroll bar's value when the user clicks in the gray area of the bar. If you want paging to increment or decrement the scroll bar's value by a specific amount, pass a non-zero value for `pageSize`.

```
wind_dirty (Windpub *x)
```

Sets the window's dirty flag, so the user will be asked to save changes if the window is closed. Your saveto method will be called if the user wants to save the changes.

```
wind_drag (methodptr dragfun, void *arg, Point start)
```

Use `wind_drag` to track mouse dragging in a window. This replaces a typical program's loop that usually looks like:

```
do {
    GetMouse(&pt);
    /* do something here */
} while (StillDown());
```

You pass a pointer to a function (`dragfun`) you want called every time the mouse moves. It will call `dragfun` with the specified argument `arg`, the location of the mouse, and whether the mouse button is down. When the mouse button goes up, your drag function is called one last time. Your drag function should be declared as follows:

```
MyDragFun (void *dragarg, Point pt, int buttonDown)
```

`dragarg` is the argument `arg` you passed to `wind_drag`. Most often it will be a pointer to your object.

Normally, `wind_drag` will only call your drag function when the mouse moves. If you want your drag function to be called regardless of whether the mouse moved, call `wind_noworrymove` before calling `wind_drag`.

```
wind_noworrymove ()
```

Sets a flag which causes `wind_drag` to pass control to your drag function even if the mouse hasn't moved. See `wind_drag` for more information.

```
wind_setcursor (int cursNum)
```

Sets the cursor to one of several predefined cursors, defined as follows:

```
#define C_ARROW        1
#define C_WATCH        2
#define C_IBEAM        3
#define C_HAND         4
#define C_CROSS        5
#define C_PENCIL       6
#define C_GROW         8
```

```
GrafPtr wind_setport (Windpub *x)
```

A convenience function that sets the current GrafPort to the port associated with the window `x`. A pointer to the previous GrafPort is returned by the function if successful, otherwise, `wind_setport` returns NIL. You should call this function before drawing or handling events in a window, and call `_SetPort` on the result when you're through.

Example:

```
GrafPtr sp;

if (sp = wind_setport(myWind)) {
    /* draw things here */
    SetPort(sp);
}
```

```
wind_setsmax (Windpub *x, int hMax, int vMax)
```

Sets the maximum values of the window's scroll bars. The minimum values are always 0.

```
wind_setsval (Windpub *x, int hVal, int vVal)
```

Sets the values of the window's scroll bars.

```
wind_settitle (Windpub *x, char *title, int brackets)
```

Sets the title of the window. If `brackets` is non-zero, the title will appear within square brackets. `title` is a C string.

```
wind_setundo (char *undoString, int undoable)
```

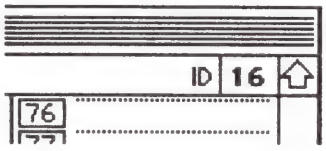
Sets the name of the Undo menu item in the Edit menu to `undoString`, which should be a C string. If `undoable` is non-zero, the Undo menu item is enabled; if `undoable` is 0, it's disabled.

```
wind_free (Windpub *x)
```

Frees the data structures associated with the Windpub `x`. Call this when your object that created the window is being freed.

Numericals

The Numerical object provides a way of displaying and editing numbers if your window. Here is an example from the window of the **lib** object:



The Numerical data structure is declared in **ext_numc.h**. You can access most of the important fields of the Numerical through functions however.

Numerical routines handle the updating of their current value and tracking the mouse. You supply a routine that gets called when the Numerical's value is changed, called the *track* routine.

```
Numerical *num_new (int top,int left,int bottom, int right,
      ProcPtr draw, ProcPtr inc, long flags, long min, long max,
      long val, int font, int fsize);
```

Creates a new Numerical object. `top`, `left`, `bottom`, and `right` are the coordinates of the Numerical's rectangle. `draw` and `inc` are routines you supply which modify the standard behavior of the Numerical. If you don't wish to supply these routines, pass `0L` for `draw` and `inc`. `draw` returns a character string for the Numerical to display instead of the standard number. It should be declared as follows:

```
void *myDrawProc (Numerical *x, long currentValue, char *dstString)
```

The draw routine should return in `dstString` a string to display based on `currentValue`.

`inc` is a routine which returns the new value of the Numerical based on the number of pixels the mouse has moved up or down while the user is scrolling. `inc` should be declared as:

```
long myIncrementProc (Numerical *x, long value, int pixelsMoved)
```

Note that `pixelsMoved` will be *negative* if the mouse has moved up since the last time the increment procedure was called. Typically, this is when the Numerical's value will *increase*. The procedure should return (as a long) the new value of the Numerical based on the arguments `value` and `pixelsMoved`. If you write an increment proc you may wish to check the new value against the `n_min` and `n_max` fields of the Numerical.

`flags` set options about the Numerical's appearance and behavior as follows:

`#define N_HILITE 1`	Numerical is highlighted, used internally
`#define N_RIGHT 2`	Text is right-justified
`#define N_LEFT 4`	Text is left-justified
`#define N_FRAME 8`	Draw a frame around the Numerical's box
`#define N_CLIP 16`	Not currently used
`#define N_ENDTRACK 32`	Call the Track Routine only when the user is finished scrolling
`#define N_BOLD 64`	Draw the text in bold

font and fsize are the font number and font size of the text drawn in the Numerical. Throughout history, most Numericals have been 9 point Geneva (font number 3).

```
num_draw (Numerical *x)
```

Call num_draw in your window's update method, and any time you want to draw the Numerical with its current value. num_draw will call your draw routine (it if exists) to set the character string to be drawn. The current GrafPort must be set to your window before calling num_draw.

```
num_hilite (Numerical *x, int way)
```

If way is 1, the Numerical will be inverted (in its hilited state). If way is 0, the Numerical will be drawn normally. For normally behaving Numericals, you won't need to use this routine.

```
num_test (Numerical *x, Point pt)
```

Returns non-zero if pt lies within the Numerical's rectangle. Call this in your window's click method. If num_test returns non-zero, call num_track.

```
num_track (Numerical *x, Point pt, ProcPtr track, void *arg)
```

Tracks the mouse and scrolls the value of the Numerical up or down. pt is the starting point (passed to your click method). track is a routine which will perform some action every time the value of the Numerical changes. If you don't pass a pointer to a function for track, there's no way to know the value of the Numerical when the mouse was released, since num_track returns immediately after you call it, and uses wind_drag (see above) to track the mouse. arg is a value which will be passed to the track routine. In most cases, this should be a pointer to your object.

The track routine should be declared as follows:

```
void *myTrackProc (myObject *x, Numerical *n, long currentValue)
```

If you set the N_ENDTRACK bit in the Numerical's flags, your track routine will only be called when the user has finished scroll the Numerical. This might be a good idea if the action you take in your track routine takes a long time, and could alter the "feel" of scrolling the Numerical. In general however, users expect that they can hold the Numerical down and observe the changes taking place in the program, so you should set N_ENDTRACK only in special cases.

```
num_setvalue (Numerical *x, long newValue, int redrawFlag)
```

Sets the Numerical to a new value. If redrawFlag is non-zero, the Numerical will redraw. Make sure you've set the current GrafPort your window if you redraw a Numerical, otherwise unpleasant results may occur.

10. Writing User Interface Objects

So far, we've talked about writing normal external objects, ones that show up in boxes with two lines at the top and bottom.

Now you'll learn the secrets to writing external objects which have a custom appearance and behavior in a Patcher window.

The normal object is just one of several objects you have to choose from in the Patcher window's palette. Others include buttons...

...sliders...

... and number boxes.

These objects have inlets and outlets just like normal objects, but they present a friendlier face to the world. You can present your own friendly face with a custom user interface object. Make sure before writing a custom user interface object that your task might not be better suited to a normal object which creates its own window. Your user interface object should enhance what a user can do within the Patcher window.

Boxes

Patcher windows are made up of two basic structures: *Boxes* and *Patchlines*. The only one of concern to you will probably be the Box.

A Box structure contains the user interface object's enclosing rectangle and a bunch of flags that determine the object's appearance and behavior. When you create a structure for your user interface object, a Box must be the first thing contained within it. Not a pointer to a Box—the Box structure itself. This structure replaces the Object that normally begins an external object structure definition. Box and other useful user interface data types and constants are defined in the Max include file **ext_user.h**.

Here's an example user interface object structure definition.

```
typedef struct myUserObject {
    Box my_box;
    long my_data1;
    long my_data2;
} MyUserObject;
```

Using this technique, the patcher window is able to treat your user interface object as a Box, even though it contains additional fields.

The All-Important SICN

How do you tell Max that your object is a user interface object? You include a SICN resource in the file that contains your code resource. You can include resources into your linked code resource file in Think C by creating a resource file with the name and folder as your project, that ends in .RSRC. The SICN should be an icon of the object's appearance. Take a cue from the size and style of the existing Max object icons (look at the SICN resources in the Max application file in ResEdit) when designing yours. Give the SICN the same resource ID number as the mAxL resource with which it's associated.

In addition, *name* the SICN resource with a descriptive name of your object, for example *Horizontal Slider*. This name will show up in the Assistance portion of the Patcher window when the user clicks on your SICN in the Patcher palette.

User Interface Object Creation Function

The `menufun` argument to `setup`, called at initialization time, points to a function that will be called if the user chooses your user interface object from the Patcher window palette. This function should be declared as follows:

```
void *myMenuFun (Patcher *p, long x, long y, long font)
```

You're passed the `x` (left) and `y` (top) location where the user wishes to place an object of your class. The `font` argument contains the current Patcher window's default font size in the lower 16 bits and font number in the upper 16 bits. It should be used if your object will display text. You should create a "default" object in size and appearance at the x,y location specified and return a pointer to it. We'll discuss how to deal with the Patcher argument in a moment.

This is not the only creation function you need to write, because your object will be created when Max files are read in, much in the same manner as normal objects are created. In this case, however, the message that gets evaluated to create you is one you wrote out yourself in a special psave method (we'll discuss this method shortly).

Typically, you'll save the coordinates of the Box (not just the left and top, since the user might shrink or expand your object's box) plus any additional information needed to duplicate the object. For example, a slider object might save its range. All of these values will be arguments to a message that will result in your object's creation function being called.

This creation function should be declared as follows:

```
myFileCreateFun (Patcher *p, Symbol *s, int argc, Atom *argv)
```

The Patcher `p` is the Patcher window in which your object is created, the Symbol `s` is the name of your object, and `argc` and `argv` refer to a traditional Max list of Atoms which hold the arguments that describe the user interface object.

At this point, you may be suffering under the burden of having to write two creation functions. However, a utility routine called `patchbox` may make life easier for you by allowing you to combine both the menu creation function and the file creation function.

```
void *patchbox (Patcher *p, methodptr creator, int argc, Atom *argv,
            int defaultWidth)
```

This is a convenience function which extracts the coordinates of a box from a list of Atoms and hands them to a creation function, `creator`, that you specify. When saving the coordinates of your user interface object, you usually save them in following order:

left,	
top,	
width	(defaults to `defaultWidth` passed to `patchbox` above),
height	(or font for text objects),
data1	(optional, defaults to 0),
data2	(optional, defaults to 0),
data3	(optional, defaults to 0)

`data1`, `data2`, and `data3` do not have to be present in the list of Atoms which is being read in.

`patchbox` will hand this list of arguments to your "real" creation function, which should be declared as follows:

```
myRealCreateFun (Patcher *p, int left, int top, int width,
            long heightOrFont, int data1, int data2, int data3)
```

This is very similar to the format that's required by the menu creation function. Here's an example that shows how `patchbox` can be used to combine the two creation functions.

```
void *myRealCreateFun();

void *myMenuFun(p, x, y, font)
Patcher *p;
int x,y,font;
{
    return (myRealCreateFun(p,x,y,DEFWIDTH,(long)DEFHEIGHT,0,0,0));
}

void *myFileCreateFun(p, s, argc, argv)
Patcher *p;
Symbol *s;
int argc;
Atom *argv;
{
    return (patchbox(p, myRealCreateFun, argc, argv, DEFWIDTH));
}
```

Now all you have to do is write one function, `myRealCreateFun`, which takes arguments as stated above.

In your "real" creation function, you'll first want to allocate memory for your object with `newobject`. Attentive programmers might be wondering: doesn't `newobject` fill in the fields of an Object, not a Box? Yes, this is true, however, the first field of the Box is in fact an Object, so everything's fine. Anyway, returning to the story, the next thing you'll do is initialize the fields of your Box, using `box_new`.

```
void *box_new (Box *b, Patcher *p, int flags, int left, int top,
      int right, int bottom)
```

`box_new` doesn't actually allocate memory for a Box, but initializes an existing Box inside your object. Pass a pointer to the Box in `b` and the Patcher your creation function was passed in `p`. `left`, `top`, `right`, and `bottom` are the coordinates of the Box. `flags` is a combination of the following constants:

```
#define F_DRAWFIRSTIN 1       /* draw first inlet */
#define F_GROWY       2       /* can grow in y direction only */
#define F_NODRAWBOX   4       /* don't draw the box  */
#define F_SPOOLED     8       /* spool out drawing the box */
#define F_DRAWN       16      /* usually never used */
#define F_GROWBOTH    32      /* can grow independently in x and y */
#define F_IGNORELOCKCLICK 64
                      /* don't send a click msg if patcher is locked */
#define F_NOGROW      128     /* don't even draw a grow thingie */
#define F_HILITE      256     /* hilitable object (for typing) */
```

The most commonly used flags are `F_DRAWFIRSTIN`, and `F_GROWY` or `F_GROWBOTH`.

After having initialized the Box, you'll want to initialize the other fields of your object. It's strongly recommended (especially given interrupt level considerations) that you use a Qelem for drawing your object. In addition, it often helps to keep track of both your object's value and its most recently drawn value. This way, you'll know when your object needs to be updated to reflect a new value.

Then create any Outlets or additional Inlets. Then assign the Box's `b_firstin` field to point to your object, as in...

```
myNewObject.my_box.b_firstin = (Inlet *)myNewObject;
```

Finally, you'll want to call `box_ready` so that the Patcher window knows to draw your newly initialized object.

```
box_ready (Box *b)
```

`box_ready` calculates where to draw your new Box's inlets and outlets. Then it visually selects the new object, which is handy if the first thing the user wishes to do is choose **Get Info...** from the Max menu to set additional parameters, such as the range for a slider object.

User Interface Object Free Function

Your free function should call `box_free` to free any data associated with the Box. When you write a Max object that isn't a user-interface object, `box_free` is called for you.

Then do any other memory disposal or cleanup as you normally would.

```
box_free (Box *b)
```

Frees data structures associated with a Box, such as patch cords. Causes the area of the Patcher window underneath the Box to be redrawn.

Additional Messages for User Interface Objects

There are three additional messages you need to implement for a user interface object: click, update, and psave. Use `addmess` at initialization time to add methods to respond to these messages. As with window messages, these messages should have argument type lists specified as:

```
A_CANT, 0
```

Here's a list of messages for user interface objects, and directions for responding to them.

The click message:

```
void *myObject_click (MyObj *x, Point pt, int modifiers)
```

The Patcher window has detected a click within your Box. You should respond by tracking the mouse in some appropriate way. `pt` is in local coordinates and modifiers is the standard Macintosh EventRecord modifiers field. Use `wind_drag` to do mouse tracking. Use `patcher_setport` to set the current GrafPort before doing any drawing or mouse tracking.

If you send any messages out your outlets in your click method, you must set the `lockout` flag before doing so, and restore it afterward. Here's an example of a click routine which sends out a 0 each time:

```
void *myObject_click(x, pt, modifiers)
MyObj *x;
Point pt;
int modifiers;
{

    int savelock;

    SetUpA4();
    savelock = lockout_set(1);
    outlet_int(x->m_ob.o_outlet,0L);
    lockout_set(savelock);
    RestoreA4();
}
```

The update message:

```
void *myObject_update (MyObj *x)
```

Draw your user interface object inside its box. The update method is where you should check to see if your object has changed in size, if this is important. For example, a slider object might want to

reconstruct a bitmap of its "slider handle" based on a new size. You'll need to store some aspect of the Box's "old" size within your object to make the comparison. Also, you can use the `box_size` function to resize a Box.

Note: it is *essential* that you call `patcher_setport` to set the current GrafPort before doing drawing of any kind. If `patcher_setport` returns `0L`, don't draw. Otherwise, your object will crash Max when it tries to draw in an invisible window, and mess things up when the Patcher window is printed. Note that since your object may function inside a hidden subpatch window, you cannot count on always being able to draw or do any kind of graphics inside your object.

The psave message:

```
void *myObject_psave (struct myObj *x, struct binbuf *dstBuf)
```

Writing the psave method is the most arcane thing you'll probably have to do in writing your user interface object. You'll usually use the function `binbuf_vinsert` (described above) to create a message that can recreate your object when a Max file is opened. First, you need to determine whether your object has been set to be hidden by checking your Box's `b_hidden` field. The user can hide any box or patchline by choosing **Hide on Lock** from the Max menu.

Then you'll save your object's internal data in a manner something like the example below. The only thing you should change from this example is the exact arguments you save, along with the format string passed to `binbuf_vinsert`.

You should also know what the name of your object's class is. Unless you find out the name of your **mAxL** code resource at initialization time and store it in a global character string, this means the name of your object gets hard coded into the program, so it must be consistent with the name you've given the mAxL resource in the Think C **Set Project Type...** dialog. In this case, our object is called **myobject** (surprised?).

Assume, for this example, that we're saving a user interface object with the following structure:

```
typedef struct myObject {
    struct box my_box;
    long my_range;
    long my_value;
} MyObject;
```

Here's the psave method:

```
void *myObject_psave (x, w)
MyObj *x;
void *w;        /* a Binbuf */
{
    int hidden;
    Rect *r;

    SetUpA4();
    hidden = myObj.my_box.b_hidden;
    r = &myObj.my_box.b_rect;

    if (hidden) {
        binbuf_vinsert (w,"sssslllll",gensym("#P"),gensym("hidden"),
                gensym("user"),gensym("myobject"), (long)(r->left),
                (long)(r->top), (long)(r->right - r->left),
                    (long)(r->bottom - r->top), (long)(x->my_range));
    } else {
        binbuf_vinsert (w,"ssslllll", gensym("#P"),gensym("user"),
            gensym("myobject"), (long)(r->left),(long)(r->top),
                (long)(r->right - r->left),
                    (long)(r->bottom - r->top), (long)(x->my_range));
    }
    RestoreA4();
}
```

A few points of explanation. hidden is a special keyword that tells Max to create the Box with its `b_hidden` attribute set. This will be done automatically for you when a Max file is opened. user is a special keyword that is required for an externally written user interface object. If you forget to include user, your object will probably generate an error something like...

```
patcher: doesn't understand 'myobject'
```

...when it is loaded from a file.

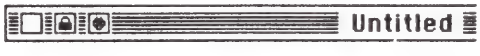
The rest of the arguments to `binbuf_vinsert` are the coordinates of your object's Box and other information you want to save.

Miscellaneous User Interface and Patcher Routines

```
int box_ownerlocked (Box *b)
```

A convenience function which returns non-zero if the Patcher window the box is contained in is locked, zero if its unlocked.

Unlocked Patcher Window



Locked Patcher Window

User interface objects may want to draw themselves differently if the Patcher window is locked. You may have noticed that a Box's outlets are extended downward one pixel when a Patcher is unlocked. This is done automatically for you.

By the way, the "owning Patcher" of a box can be found in the Box's `b_patcher` field.

```
box_size (Box *b, int hSize, int vSize)
```

Resizes the Box's rectangle to be of width `hSize` and height `vSize`;

```
box_nodraw (Box *b)
```

Returns non-zero if the Box's Patcher is locked and the Box is hidden (because the user has chosen **Hide on Lock** from the Max menu). In this case, your object should not draw in response to messages it receives. For example, sliders normally draw their value in response to int messages, but should check `box_nodraw` before doing so.

```
box_redraw (Box *b)
```

Erases and redraws the entire Box as well as any other Boxes or Patchlines in the affected area. This is useful when changing something in the Box that isn't normally redrawn, such as the maximum range of a slider.

```
patcher_deselectbox (Patcher *p, Box *b)
```

Visually de-selects a highlighted Box `b` contained in the Patcher `p` (if the Box is highlighted).

```
patcher_dirty (Patcher *p)
```

Sets the dirty (modified) bit in a Patcher, so that the user will asked to save changes if the window is closed. The only time you have to do this is if your object stores its data inside its owning patcher (for example, a **table** object can store its elements inside a Patcher).

Note: If your object is not a user interface object, you can find your Box's "owning patcher" in your object creation function (this is the *only* place this will work) using the technique in the following code fragment.

```
void *mypatcher;

mypatcher = ((Symbol *)gensym("#P"))->s_thing;
```

```
patcher_selectbox (Patcher *p, Box *b)
```

Visually selects a highlighted Box `b` contained in the Patcher `p` (if the Box isn't currently highlighted). This is done for you when you call `box_ready` in your creation function.

```
GrafPtr patcher_setport (Patcher *p)
```

A convenience function which sets the current `GrafPort` to the Patcher's `GrafPort`. `patcher_setport` returns the previous port, or `0L` if an error occurs (for example, if the patcher window isn't visible—definitely check for this before drawing!). After you're through drawing in the patcher window, call `_SetPort` with the result of a successful `patcher_setport`. Note that this function is *not* equivalent to `wind_setport`, though the way you'd use it is similar to the example shown under `wind_setport` above. Do not use `wind_setport` instead of `patcher_setport` in a User Interface object.

```
patcher_okclose (Patcher *p, Object *target)
```

After calling `patcher_okclose`, the Patcher window will send any okclose message it receives on to the object `target`. In the case where your object opens a Patcher window itself, you might want to handle how the window is saved or closed in a non-standard way. You should then write an okclose method and pass yourself to `patcher_okclose` after you've opened the window.

```
ispatcher (Object *x)
```

Returns non-zero if object `x` is a Patcher, zero if not. You might use this to locate all the patcher windows from the window list (use `_GetWRefCon` to get a pointer to the Max Windpub object associated with a Macintosh window, then look in the `w_assoc` field of the Windpub).

Or, if you're traversing through the list of a Patcher's boxes, you might be interested to know if an object associated with the Box (in the `b_firstin` field) is a subpatcher. Don't pass the Box itself, rather check to see if `b_firstin` is non-zero, and if so, pass it to `ispatcher`. It's important to make sure you pass a genuine Max object to `ispatcher`.

```
isnewex (Object *x)
```

Returns non-zero if object x is a Box that holds the text of normal objects. You can then find the pointer to the object itself in the Box's `b_firstin` field. Make sure you pass a genuine Max object to `isnewex`.

Note: You can always find the type of an object using the macros `ob_name` or `ob_class` defined in **ext_mess.h**, then comparing the string (in the case of `ob_name`) or symbol (in the case of `ob_class`) to the one you're looking for. `ispatcher` and `isnewex` are just a bit faster.

```
newex_knows (Box *b, Symbol *msg)
```

Returns non-zero if the Box `b` is a Box that holds the text of normal objects and its associated object has a method for the Symbol `msg`.

```
patcher_eachdo (fptr fun, void *arg)
```

This function calls the function `fun` for every Patcher in memory, including hidden subpatchers. `fun` should be declared as follows:

```
myEachFun (patcher, arg)
struct patcher *patcher;
void *arg;
{

}
```

`arg` is some argument you want passed to `fun` (in addition to the Patcher). Your function needs to return 0 if you wish to continue being called for each Patcher, non-zero if you want to stop. You might use `patcher_eachdo` to implement a search routine for a particular object. It is used by the built-in objects **send** and **receive** in response to a dblclick message. These objects search all Patchers for any other **send** or **receive** objects bound to the same Symbol, and bring the corresponding object's window to the front.

User Interface Objects with Text

If you want to write a user interface object which allows text to be typed into it (like the **comment**, message box, or object box), the routines described in this section will be helpful.

Text boxes maintain both a Binbuf, which contains the text in binary form, and a TextEdit record, which shows the text as characters that can be edited.

Note that if you want your object to *display* text but not edit it, it's easier to use `binbuf_new` (to create an "independent" Binbuf) than to use these routines, which bank heavily on the Binbuf/TextEdit record duality.

```
void *textbox (Patcher *p, methodptr f, int argc, Atom *argv)
```

`textbox` is analogous to `patchbox` for formatting calls to user interface object creation functions. It calls function `f` with the first four arguments in the `argv` list as integers (assuming that they're the `left`, `top`, `width`, and `font` of the text Box). Any additional arguments are passed in `argc`, `argv` format. A creation function to be called by `textbox` would have arguments declared as follows:

```
void *myCreateFun (Patcher *p, int left, int top, int width, long font,
        int argc, Atom *argv)
```

Presumably, the additional arguments in `argv` would be the "text" that goes in the newly created box. To actually make it possible to type text into the Box, call `box_binbuf` to create a Binbuf for the Box, followed by `box_settext` to set the text of the box from the arguments your creation function received.

To be consistent with other text objects, the `font` argument will contain the font number in the upper 16 bits and the font size in the lower 16 bits.

```
void *box_binbuf (Box *b, int inset, int fontSize)
```

Creates a Binbuf and TextEdit record for a Box. `inset` is the distance you want the text to appear from the top of the box. `fontSize` should be the `fontSize` parameter you were passed.

```
void *box_settext (Box *b, int argc, Atom *argv)
```

Sets the Binbuf of the Box to contain the Atoms in `argv`. Sets the `TextEdit` record of the box to contain the text from the Binbuf. If you haven't created the Binbuf, an error will be displayed.

```
void *binbuf_savebox (Binbuf *boxbuf, Binbuf *dst, Symbol *boxClass,
     int left, int top,int width, long font, int hidden, int user)
```

This is a convenience function which saves a box that has created a Binbuf (it uses `binbuf_vinsert`). It's usually called in a user interface object's psave method. `boxbuf` is the Box's own Binbuf to be saved. `dst` is the destination Binbuf, which will be passed as an argument to your psave method. `boxClass` is a Symbol which is the name of your user interface object's class. `left`, `top`, `width` are the dimensions of the box. `font` is a longword description of the Box's current font and font size as described above. `hidden` should be the value of the Box's `b_hidden` field. `user` is always 1 for external user interface objects.

```
t_cpytext ()
```

Causes Max to call `_TEToScrap` on the next suspend event. Use `t_cpytext` if the text in your box has been copied to the TextEdit scrap with a Cut or Copy command.

11. Graphics Windows

Max contains a set of routines for managing bitmap graphics and offscreen "sprites" needed to do animation and paint-program interaction. Graphics windows, known as GWinds, are created by the Max user with the **graphic** object, but they can also be created directly from your external object by calling `gwind_new`. Graphics windows are primarily display vehicles—if you want to create a specialized user interface, you're better off creating your own window (see Chapter 9—Doing Windows). Max programmers can access the mouse in a graphics window using the **MouseState** object, but a typical object that uses a graphics window will draw something—animation, shapes, text, etc.

The GWind data structure and pertinent constants are declared in **ext_anim.h**.

Graphics Window Routines

```
colorinfo (CInfoRec *cinfo)
```

`colorinfo` fills in the fields of a previous allocated CInfoRec, which is defined as follows:

```
typedef struct {
    int hasColorQD;
    int depth;
    int has32bitQD;
    int inColor;
    int curDevH;
    int curDevV;
} CInfoRec;
```

`hasColorQD` is non-zero if the machine has Color Quickdraw in ROM. `depth` is the bit depth of the screen (1-24). `has32bitQD` is non-zero if the system has 32-bit Quickdraw installed. `inColor` is non-zero if there are colors, 0 if the monitor is set to display gray scales. `curDevH` and `curDevV` are width and height, respectively, of the current GDevice.

Typcially, you'll call `colorinfo` in your Initialization routine, and store the results in global variables. Note that `depth` and `inColor` could change over time. If these are important to you, consider calling `colorinfo` more regularly.

```
GWind *gwind_new (Object *assoc, Symbol *s, int styleFlags,
      int left, int top, int bottom, int right)
```

Creates a new GWind object. `assoc` should be a pointer to your object (which will own the GWind and free it using `freeobject` when the owner dies). `s` should be a Symbol if you want other objects to be able to access the GWind. All access of GWinds is done through binding to Symbols, similar to how the Max **table** object is bound to a Symbol. If you wish your window to be without a border and title bar, pass 1 for `styleFlags`. Other flags may be defined in the future.

`left`, `top`, `bottom`, and `right` are the coordinates of the graphics window in global coordinates. Unlike `wind_new`, where you have a choice, all graphics windows you create are immediately visible.

```
gwind_offscreen (GWind *x)
```

Initializes an Offscreen record for the graphics window `x`. You can also send the offscreen message to a GWind to call this function.

```
GWind *gwind_get (Symbol *s)
```

If a GWind object is associated with the Symbol `s`, `gwind_get` returns a pointer to it. Otherwise, it returns 0. You should call `gwind_get` every time one of your methods wants to start accessing a GWind, because you never know whether the GWind object still exists or not. See the example under `gwind_setport` for a typical use of `gwind_get`.

```
GrafPtr gwind_setport (GWind *x)
```

If the GWind `x` is visible, `gwind_setport` does a `_SetPort` to its associated Macintosh window and returns the previous GrafPort. If the window is not visible, `gwind_setport` returns 0. All drawing into a graphics window should be prefaced by a call to `gwind_setport`, as in the example below. Note also the correct use of `gwind_get`, assuming the `m_windsym` field of myObject is a Symbol which (supposedly) holds the name of a graphics window object.

```
myDrawMethod(x)
myObject *x;
{
    GrafPtr save;
    GWind *g;

    SetUpA4();
    if (g = gwind_get(x->m_windsym)) {    /* does it exist? */
        if (save = gwind_setport(g)) {    /* is it visible? */
            /* draw something in the GWind here */
            SetPort(save);
        }
    }
    RestoreA4();
}
```

Offscreen Routines

Although GWinds "know" about the offscreen and sprite routines, there's no reason why you can't use them in your own window if you wish. As mentioned above, `gwind_offscreen` will initialize an offscreen structure for a GWind. You can initialize the offscreen structure for your own window with `off_new`.

The Offscreen routines take care of "buffering" drawing to minimize unsightly screen flicker. This facility is similar to that provided by 32-bit Quickdraw GWorld functions, and the Offscreen structure transparently "uses" 32-bit Quickdraw if it is the system. The user of the routines does not need to worry about whether GWorlds are being used or not.

Typically, you'll use the Sprite routines to draw into an Offscreen structure.

```
Offscreen *off_new (GrafPtr destination)
```

Creates a new Offscreen structure which will have the same dimensions as the GrafPort it is linked to. If the size of your window changes, call `offscreen_resize`. You must create an Offscreen before you can create or use any Sprites.

```
off_free (Offscreen *discard)
```

Frees the memory associated with an Offscreen structure.

```
off_copy (Offscreen *src)
```

Copies the entire contents of src to its destination GrafPort.

```
off_copyrect (Offscreen *src, Rect *srcRect)
```

Copies `srcRect` in `src` to the same location in `src`'s destination GrafPort.

```
off_maxrect (Offscreen *o, Rect *src1, Rect *src2, Rect *dest)
```

Utility routine which returns in `dest` a rectangle which covers the part of both `src1` and `src2` that lies completely inside the Offscreen's bounding rectangle.

```
off_tooff (Offscreen *dst, PixMapHandle *src, Rect *srcRect,
     Rect *dstRect)
```

Copies the pixels in `src` to the Offscreen buffer without copying them to the screen. `src` can also be a pointer to a BitMap on a non Color Quickdraw system. `srcRect` and `dstRect` are the source and destination rectangles of the copy.

```
off_resize (Offscreen *x)
```

Resizes the area of the Offscreen buffer to match its current associated GrafPort. Call this routine when the user resizes a window containing an Offscreen.

Sprite Routines

Sprites are independent "creatures" that draw some arbitrary stuff inside a set rectangle. A Sprite "system" is owned by an Offscreen object, so for example, there's a different set of Sprites for each active Graphics Window in Max. Each Sprite has a *priority*, which is used by the system to "layer" objects from front to back. Sprites can change their priority dynamically. There is no set limit to the number of Sprites or different priority levels. If two or more Sprites are at the same priority level, the one which joined the Offscreen structure first will be drawn in front of the more recent arrivals.

The Sprite data structure is defined in **ext_anim.h**. You can get away with being ignorant of the fields of a Sprite object, but it be helpful in some circumstances, such as knowing the Sprite's rectangle.

```
Sprite *sprite_new (Object *assoc, Offscreen *owner, long priority,
          Rect *sRect, ProcPtr *drawProc)
```

Creates a new Sprite object that will draw in the Offscreen environment of `owner`. `assoc` should be a pointer to your object, it will be passed to your `drawProc` routine. `priority` is the sprite's priority, where 0 is the "background" and higher priority numbers are more in the foreground. `sRect` is the rectangle your Sprite will draw inside of. It can be moved and changed after the Sprite has been created. `drawProc` is your function that will draw the Sprite based on its current state. It should be declared as follows:

```
myDrawProc (x, s)
MyObject *x;
Sprite *s;
{

}
```

All of the sprite drawing routines discussed below automatically redraw the other sprites on the screen if necessary when the Offscreen owner of the Sprite's destination GrafPort is visible.

```
sprite_move (Sprite *s, int deltaH, int deltaV)
```

Move a Sprite's rectangle by `deltaH` pixels horizontally and `deltaV` pixels vertically. The Sprite is erased at its old location and redrawn at the new location. Any other Sprites affected by the move are also redrawn. If you want to redraw your sprite at the same location but with a different appearance, you could use:

```
sprite_move(mySprite, 0,0);
```

```
sprite_moveto (Sprite *s, int h, int v)
```

Move the Sprite's rectangle to location `h` (horizontal), `v` (vertical). The sprite is erased at its old location and redrawn at the new location. Any other Sprites affected by the move are also redrawn.

```
sprite_rect (Sprite *s, Rect *newRect, int change, int next)
```

Change the Sprite's rectangle to `newRect` and redraw. `change` and `next` are values to store in the change and next fields of the Sprite. These fields can be interpreted by your Sprite's drawProc in any way it wants.

```
sprite_redraw (Sprite *s, int deltaH, deltaV, int change, int next)
```

Like `sprite_move`, but `sprite_redraw` allows you to set the change and next fields of the Sprite.

```
sprite_erase (Sprite *s)
```

Erases the Sprite, filling in any other Sprites which may have been lurking behind it.

```
sprite_newpriority (Sprite *s, long new)
```

Assigns the value `new` to the Sprite's priority. Redraws the Offscreen structure that owns the Sprite to reflect the change in priorities.

A Sprite Example

The following example includes key methods of an object that draws ovals using Sprites. It shows several useful techniques, such as isolating the data structures used for drawing in a Qelem from those changed inint methods.

The **Oval** object has four inlets, for each coordinate of an oval's rectangle. The **Oval** data structure is defined as follows:

```
typedef struct oval {
    struct object o_ob;
    long o_priority;        /* sprite priority */
    Sprite *o_sprite;       /* the sprite */
    Rect o_bounds;          /* where it is */
    Rect o_dbounds;         /* where it is drawing */
    void *o_qelem;          /* Qelem */
    Symbol *o_sym;          /* symbol of GWind */
} Oval;
```

Here's the Initialization routine:

```
fptr *FNS;
void *OvalClass;
void *oval_new(),*oval_int(),*oval_in1(),*oval_in2(),*oval_in3(),
    *oval_free(),*oval_qfn(),*oval_bang();

main(f)
fptr *f;
{
    RememberA0();
    SetUpA4();
    FNS = f;

    setup(&OvalClass, oval_new, oval_free, sizeof(Oval), 0L, A_SYM,
        A_LONG, A_DEFLONG, A_DEFLONG, A_DEFLONG, A_DEFLONG, 0);
    addint(oval_int);
    addinx(oval_in1,1);
    addinx(oval_in2,2);
    finder_addclass("Graphics","oval");
    RestoreA4();
}
```

Here is the creation function for **Oval**. Note that its Sprite isn't created until it's needed, within the queue function `oval_qfn`. We always reference the GWind through a Symbol each time we draw, since we have no guarantee the GWind is still around when we want to draw in it.

```
void *oval_new(windName, priority, left, top, bottom, right)
Symbol *windName;
long priority, left, top, bottom, right;
{
    void *qelem();
    struct oval *x;
    SetUpA4();
    x = (struct oval *)newobject(OvalClass);
    intin(x,3);
    intin(x,2);
    intin(x,1);
    SetRect(&x->o_bounds,(int)left,(int)top,(int)bottom,(int)right);
    x->o_dbounds = x->o_bounds;
    x->o_sym = windName;
    x->o_priority = priority;
    x->o_qelem = qelem_new(x,oval_qfn);
    x->o_sprite = 0;
    RestoreA4();
    return (x);
}
```

Here are the int methods. These set the coordinates of the `o_bounds` rectangle and the leftmost one sets the Qelem to draw the oval. You can't draw on the screen directly in response to an int or bang message, since your method may be executing at interrupt level.

```
void *oval_bang(x)
struct oval *x;
{
    SetUpA4();
    qelem_set(x->o_qelem);
    RestoreA4();
}

void *oval_int(x,left)
struct oval *x;
long left;
{
    x->o_bounds.left = left;
    oval_bang(x);
}

void *oval_in1(x,top)
struct oval *x;
long top;
{
    x->o_bounds.top = top;
}

void *oval_in2(x,right)
struct oval *x;
long right;
{
    x->o_bounds.right = right;
}
```

```
void *oval_in3(x,bottom)
struct oval *x;
long bottom;
{
    x->o_bounds.bottom = bottom;
}
```

The queue function is where all the action is.

```
void *oval_qfn(x)
struct oval *x;
{
    GrafPtr gp;
    GWind *it;

    SetUpA4();

    x->o_dbounds = x->o_bounds;    /* make a copy of the new rect */
    it = gwind_get(x->o_sym);

    if (!it || !(gp = gwind_setport(it))) {
                                   /* doesn't exist or not visible? */
        RestoreA4();
        return;
    }
    if (it->g_off) {               /* if there's an Offscreen */
        if (!x->o_sprite)          /* make a new Sprite? */
            x->o_sprite = sprite_new(x,it->g_off,x->o_priority,
                &x->o_dbounds,oval_spritedraw);

        sprite_rect(x->o_sprite,&x->o_dbounds,0,0);      /* draw */  }
    SetPort(gp);
    RestoreA4();
}
```

Here is the Sprite's `drawProc`. Note that we get the drawing bounds from the Sprite's rectangle `s_rect`. This isn't actually necessary, but ensures that you'll always be drawing where the Sprite thinks you're supposed to be drawing.

```
void *oval_spritedraw(x,s)
Oval *x;
Sprite *s;
{
    SetUpA4();
    x->o_dbounds = s->s_rect;
    PaintOval(&x->o_dbounds);
    RestoreA4();
}
```

Finally, here's the free function.

```
void *oval_free(x)
struct oval *x;
{
    SetUpA4();
    qelem_free(x->o_qelem);
    if (x->o_sprite)
        freeobject(x->o_sprite);
    RestoreA4();
}
```

Hopefully this example demonstrates how simple it is to draw things in GWinds with Sprites. The actual Max **oval** object is a bit more complicated than this one (it can draw in different shapes and colors) but the Sprite techniques it uses are identical to those presented in this example.

12. Writing External Functions for expr

Max's **expr** object is a powerful variable expression evaluator which uses C-like syntax. It is possible to define new C functions that can be used inside an **expr** using externally loaded code resources. These code resources are similar to the external object definition described in previous chapters, but have a somewhat simpler interface. Note: You might want to become more familiar with using **expr** inside Max before reading this section.

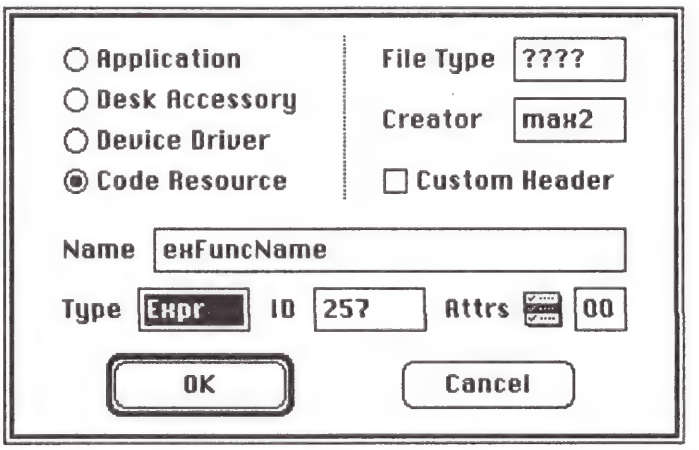
Suppose we wanted to write an external function timeofday which accessed the Macintosh's clock and returned the time of day in seconds since midnight. Here's an example of something you might type into an **expr** to access this capability. (As examples of **expr** clients, note that **expr** is used in the **if** object and in is statements in the **lib** object's script language.)

expr works by parsing the string it's handed—in this case timeofday(0)—and looking up the name of the function which precedes a set of parenthesis. Currently, all parenthesis must enclose at least one token, so if your function doesn't need to take any arguments (as would be the case with timeofday), you need to declare it as taking one argument, which you can ignore. Unlike standard Max argument lists defined with `A_GIMME`, **expr** functions cannot take a variable number of arguments.

What To Do In Think C

In the Set Project Type dialog, define your project as a Code Resource, file type **????** and creator **max2**. Assign your code resource type **Expr**, and then choose the resource ID so that the lower four bits indicate the number of arguments your function expects. This means, essentially, take some number, multiply it by 16, and add the number of arguments. Remember, you need at least one argument.

The name of your code resource is the name of the function inside **expr**. Below, we're defining a function called exFuncName which takes one argument.



In your source file, you need to include **ext_expr.h**, and if you wish to use any Max function macros, you'll need to include **ext.h** and **SetUpA4.h**.

Types and Declarations

expr uses some special data structures to communicate with your function. The arguments and result are of type Ex_ex, similar (but not identical) to an Atom. Ex_ex is defined as follows:

```
typedef struct ex_ex {
    union {
        long v_int;
        float v_flt;
        long op;
        char **ptr;
    } ex_cont;          /* content */
    long ex_type;       /* type of the node */
} Ex_ex;
```

The following macros are used to shorten the access for particular elements:

```
#define     ex_int      ex_cont.v_int
#define     ex_flt      ex_cont.v_flt
#define     ex_op       ex_cont.op
#define     ex_ptr      ex_cont.ptr
```

You'll need to check the `ex_type` field of an Ex_ex to determine the type of the element, then use the proper access macro. The possible types are defined as follows:

```
#define     ET_INT      0x1         /* an int */
#define     ET_FLT      0x2         /* a float */
#define     ET_OP       0x3         /* operator */
#define     ET_STR      0x4         /* string */
#define     ET_TBL      0x5         /* a table, the content is a pointer */
#define     ET_FUNC     0x6         /* a function */
#define     ET_SYM      0x7         /* symbol ("string") */
#define     ET_VSYM     0x8         /* variable symbol ("$s?") */
```

For almost every situation, you'll be dealing with `ET_INT` and `ET_FLT` types. You should probably do nothing (or return an error) rather than deal with a type other than what you're expecting. It's unlikely that you'll be passed anything but integers and floats (in fact, the implementation of **expr** would have to change radically before this will happen).

Declare your function as:

```
long main (argc, argv, optr)
long argc;
Ex_ex *argv,*optr;
{

}
```

`argc` is the number of arguments you'll receive (which you specified by numbering your **Expr** resource). Notice that unlike other Max calls, it is passed as a long. `argv` is a pointer to an array of the arguments passed to your function (from left to right), and `optr` is a pointer to an Ex_ex where you should place the result of evaluating your function.

Initialization

When it is installed, your function is called with `argc` set to -1 (it will be always be positive at other times). For this case only, `argv` points to the table of Max external support functions, the same as is passed to an external object initialization function. In addition, `optr` is a pointer to a Macintosh SysEnvRec returned from `_SysEnvirons`, so you can find out about the system you're running on before doing any machine-specific processing, such as accessing the floating point coprocessor. You can be sure that when `argc` is -1, you are not being called at interrupt level.

Note that at other times, your function may definitely be called at interrupt level.

You will want to access Max support macros if you want to print error messages (use `post` or `error`) or allocate memory in an interrupt (using `getbytes` and `freebytes`).

The following code fragment shows how to access Max support macros from your function.

```
#include "ext.h"
#include "ext_expr.h"
#include "SetUpA4.h"

fptr *FNS;

long main (argc, argv, optr)
long argc;
Ex_ex *argv,*optr;
{
    if (argc == -1) {
        RememberA0();
        SetUpA4();
        FNS = (fptr *)argv;

        /* other initialization here */

        RestoreA4();
        return;
    }

    SetUpA4();

    /* normal function operation goes here */

    post("thank you for calling my function!");
        /* A Max macro being called */

    RestoreA4();
}
```

Performing Computation

In general, your expr function may perform different computations depending on the types of its arguments. First, you'll check the `ex_type` field of your arguments, to see if each one is an `ET_INT` or an `ET_FLT`. Then perform your computation and set the type and value of `optr` with the result. If your arguments are both `ET_FLT`, you should probably return `ET_FLT` as a result, even if your function normally deals with integers.

The following example (which is already provided as a built-in **expr** function min) computes the minimum of two numbers. Just to keep things simple, it doesn't perform any initialization, use any global variables, or access Max function macros. It *does* show how to access and type-check function arguments and return a result.

```
long main(argc, argv, optr)
long argc;
register Ex_ex *argv, *optr;
{
    register Ex_ex *left, *right;

    if (argc == -1)    /* REQUIRED check for initialization time */
        return;

    left = argv++;
    right = argv;
    if (left->ex_type == ET_FLT) {
        if (right->ex_type == ET_FLT) {
            if (left->ex_flt < right->ex_flt)
                *optr = *left;     /* copy arg -> result, incl. type */
            else
                *optr = *right;
        } else {
            if (left->ex_flt < (float)right->ex_int)
                *optr = *left;
            else
                *optr = *right;
        }
    } else {
        if (right->ex_type == ET_FLT) {
            if ((float)left->ex_int < right->ex_flt)
                *optr = *left;
            else
                *optr = *right;
        } else {
            if (left->ex_int < right->ex_int)
                *optr = *left;
            else
                *optr = *right;
        }
    }
}
```

The following example calls the built-in C function pow(), converting its arguments to double before calling the function, and converting the result of pow back to an integer if both arguments are integers, or float otherwise. Note that you would have to include the Math library in your code resource to call pow(). Since all of the Math library's functions are already available in **expr** anyway, this is a pretty hypothetical example.

```
long main(argc, argv, optr)
long argc;
register Ex_ex *argv, *optr;
{
    register Ex_ex *left, *right;

    if (argc == -1)
        return;

    left = argv++;
    right = argv;
    optr->ex_type = ET_FLT;        /* set output type */

    if (left->ex_type == ET_FLT) {
        if (right->ex_type == ET_FLT) {
            optr->ex_flt = (float)
                pow((double)left->ex_flt,(double)right->ex_flt);
        } else {
            optr->ex_flt = (float)
                pow((double)left->ex_flt,(double)right->ex_int);
        }
    } else {
        if (right->ex_type == ET_FLT) {
            optr->ex_flt = (float)
                pow((double)left->ex_int,(double)right->ex_flt);
        } else {
            optr->ex_type = ET_INT;
            optr->ex_int = (int)
                pow((double)left->ex_int,(double)right->ex_int);
        }
    }
}
```

13. Index of Function Macros

addbang 20
addfloat 20
addftx 20
addint 20
addinx 21
addmess 21
alias 10
assist_string 39
bangout 24
binbuf_append 30
binbuf_eval 30
binbuf_free 32
binbuf_getatom 30
binbuf_insert 30
binbuf_new 30
binbuf_read 32
binbuf_savebox 72
binbuf_set 31
binbuf_text 31
binbuf_totext 31
binbuf_vinsert 30
binbuf_write 32
box_binbuf 71
box_free 66
box_new 65
box_nodraw 69
box_ownerlocked 68
box_ready 65
box_redraw 69
box_settext 72
box_size 69
clock_delay 32
clock_free 34
clock_getextfmt 33
clock_new 32
clock_set 33
clock_unset 33
clock_xdelay 33
clock_xset 33
clock_xunset 33
colorinfo 73
defer 37
defer_low 37
defvolume 40
disposhandle 40
drawstr 40
ed_free 45
ed_new 45
ed_settext 45
ed_vis 45
egetfn 43
error 38
evnum_get 49
evnum_incr 49
expr_eval 46
expr_free 46
expr_new 46
finder_addclass 11
floatin 23
floatout 24
freebytes 40
freeobject 39
gensym 37
getbytes 40
getexttime 33
getfn 42
gettime 33
growhandle 40
gwind_get 74
gwind_new 73
gwind_offscreen 74
gwind_setport 74
inlet_new 23
intin 23
intout 25
isnewex 70
ispatcher 70
isr 37
listout 25
locatefile 41
locatefiletype 41
lockout_set 36
maxversion 39
midiinfo 49
newex_knows 70
newhandle 40
newinstance 42
newobject 22
num_draw 61
num_hilite 61
num_new 60
num_setvalue 61
num_test 61
num_track 61
off_copy 75
off_copyrect 75
off_free 75
off_maxrect 75
off_new 75
off_resize 75
off_tooff 75
open_dialog 40
ouchstring 37
outlet_anything 27
outlet_bang 27
outlet_float 27
outlet_int 28
outlet_list 28
outlet_new 25
patchbox 64
patcher_deselectbox 69
patcher_dirty 69
patcher_eachdo 70
patcher_okclose 70
patcher_selectbox 69
patcher_setport 70
post 38
postatom 38
preset_int 48
preset_set 48
preset_store 47
qelem_free 36
qelem_front 35
qelem_new 35
qelem_set 35
qelem_unset 35
rescopy 12
saveas_dialog 41
serialno 41
setup 9
sprintf 39
sprite_erase 76
sprite_move 76
sprite_moveto 76
sprite_new 76
sprite_newpriority 77
sprite_rect 76
sprite_redraw 76
sscanf 39
t_cpytext 72
table_dirty 44
table_get 43
textbox 71
typedmess 42
wind_defaultscroll 58
wind_dirty 58
wind_drag 58
wind_free 59
wind_invis 58
wind_new 57
wind_noworrymove 58
wind_setcursor 59
wind_setport 59
wind_setsmax 59
wind_setsval 59
wind_settitle 59
wind_setundo 59
wind_vis 57
zgetfn 43

Output incoming MIDI pitch bend values

bendin

See Also

bendout	Transmit MIDI pitch bend messages
midiin	Output incoming raw MIDI data
xbendout	Prepare extra precision MIDI pitch bend messages
xbendin	Interpret extra precision MIDI pitch bend values
Ports	How ports are specified
Tutorial 16	More MIDI ins and outs

bendout

Transmit MIDI pitch bend messages

Input

int — In left inlet: The number is transmitted as a MIDI pitch bend value on the specified channel and port. Numbers are limited between 0 and 127.

In right inlet: The number is stored as the channel number on which to transmit the pitch bend messages.

float — Converted to int.

list — In left inlet: The first number is the pitch bend value, and the second number is the channel, of a MIDI pitch bend message, transmitted on the specified channel and port.

port — In left inlet: The word port, followed by a letter a-f, specifies the port used to transmit the MIDI messages.

Arguments

a-f — Optional. Specifies the port for transmitting MIDI pitch bend messages. Channel numbers greater than 16 received in the right inlet will be *wrapped around* to stay within the 1-16 range. If there is no argument, **bendout** initially transmits out port a, on MIDI channel 1.

a-f and int — A letter and number combination (separated by a space) indicates a port and a specific MIDI channel on which to transmit pitch bend messages. Channel numbers greater than 16 will be *wrapped around* to stay within the 1-16 range.

int — A number alone can be used in place of a letter and number combination. Numbers 1-16 specify a channel number on port a, numbers 17-32 specify channels 1-16 on port b, etc. In this case, channel numbers received in the right inlet will specify both the port and the channel number.

If the MIDI Manager is being used, a letter a-f specifies a virtual port. If the MIDI Manager is not being used, the letter a specifies the modem port and b specifies the printer port.

Output

(MIDI) — There are no outlets. The output is a MIDI pitch bend message transmitted directly to the serial ports, or to the *virtual ports* of the MIDI Manager.

Send a bang to many places, in order

bangbang / b

Input

anything — Causes a bang to be sent out all outlets, in right-to-left order.

Arguments

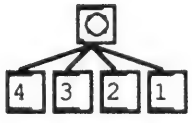
int — Optional. Sets the number of outlets. Limited between 1 and 10. Any number greater than 10 is set to 10; any number less than 1 is set to 2. If there is no argument, there will be 2 outlets.

float — Converted to int.

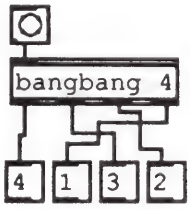
Output

bang — When a message is received in the inlet, bang is sent out each outlet, in order from right to left.

Examples



Order is normally right-to-left



Order is specified by **bangbang**

See Also

button — Flash on any message, send a bang
trigger — Send a number to many places, in order
Tutorial 7 — Right-to-left order

bendin

Output incoming MIDI pitch bend values

Input

(MIDI) There are no inlets. The input is a MIDI pitch bend message received directly from the serial ports, or from the *virtual ports* of the MIDI Manager.

Arguments

a-f Optional. Specifies the port from which to receive incoming pitch bend messages. If there is no argument, **bendin** receives from all channels on all ports.

a-f and int A letter and number combination (separated by a space) indicates a port and a specific MIDI channel on which to receive pitch bend messages. Channel numbers greater than 16 will be *wrapped around* to stay within the 1-16 range.

int A number alone can be used in place of a letter and number combination. Numbers 1-16 specify a channel number on port a, numbers 17-32 specify channels 1-16 on port b, etc.

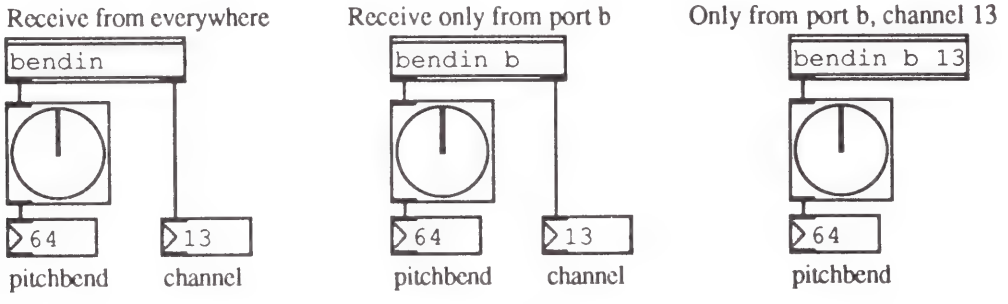
If the MIDI Manager is being used, a letter a-f specifies a virtual port. If the MIDI Manager is not being used, the letter a specifies the modem port and b specifies the printer port.

Output

int If a specific channel number is included in the argument, there is only one outlet. The output is the incoming pitch bend value from 0-127 (the most significant byte of the MIDI pitch bend message) on the specified channel and port.

If there is no channel number specified by the argument, **bendin** will have a second outlet, on the right, which will output the channel number of the incoming pitch bend message.

Examples



Pitch bend messages can be received from everywhere, a specific port, or a specific port and channel

Considerations for aspatches Mode

aspatches mode loads or dumps an entire bank as a set of individual patch requests or dumps. In this mode, the outbank and inbank statements are designed to get a single patch. These statements use the patchnum sub-statement to specify which patch is being loaded or received. The **lib** object will send each message once for each patch in a bank. As an example, consider the PCM70, which only sends and receives one patch at a time. This is the outbank statement used in the PCM70 script.

```
#L outbank 0xF0 0x06 0 0x19 / patchnum / 0x02 0x4E /
          data 334 / checksum 1 / 0xF7;
```

When the **lib** object receives the dump message, the value of patchnum will range from 0 to 49 (there are 50 patch locations in the PCM70). In addition, **lib** will insert a short delay (100ms) between each message.

Overview of Writing Editors

The **lib** object has three "friends" which make it possible to connect a patcher to communicate with a library. Using the editor keyword in the script file, you can define a Max patcher file as an editor. When you double-click on the number of a patch, the editor window is loaded and opened. The patcher must be located in the same folder as the patcher which contains the parent **lib** object, or in another folder in Max's search path.

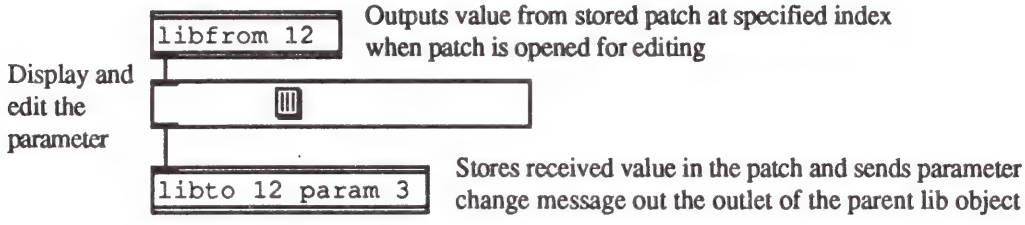
You use **libfrom** objects to get values from a patch in the library when the editor window opens and **libto** objects to put values back in the patch and (optionally) send parameter change messages. The **libswitch** object can be used to switch from one patch to another inside an editor window.

When creating an editor, we suggest the following procedure:

- Get your **lib** loading and dumping patches properly, and save a file with a bank of patches in it.
- Save an empty patcher with a name like myDeviceEdit.
- Add the param keyword followed by a parameter change message to your script, if the device accepts parameter changes. If the parameter is specified by its index into a patch, use $i4 to refer to the parameter "number" otherwise use $i1 and remember to use param and the proper parameter number in the arguments to each **libto** object you create.
- Add a statement with the editor keyword followed by the name of the saved empty patcher to your script
- Open or load a bank of patches, and double-click on one to open the editor patcher
- Unlock the window and begin creating the editor. The **libto**, **libswitch**, and **libfrom** objects you create will actually work in the proper context. If you created the patcher in isolation from its librarian, these objects will be disabled, and you won't be able to test your work as go, which is one of the benefits of using Max to develop an editor in the first place.

Note that **libfrom** and **libto** objects can be placed in subpatchers of the patcher named in the editor statement in the script file, but **libswitch** objects (which change the patch being edited) will only work if they are located in the top level Patcher window of your editor.

For each parameter you want to edit, you will need a configuration like the three objects shown below:


libfrom 12
Outputs value from stored patch at specified index when patch is opened for editing
Display and edit the parameter
libto 12 param 3
Stores received value in the patch and sends parameter change message out the outlet of the parent lib object

Example: An Entire Script File

This is a script for a librarian that deals with Programs for the Lexicon PCM70, which requires the aspatches style of loading and sending banks of patches. For best results, aspatches should be defined above any of the MIDI Communication statements.

```
#L define pcm70 PCM 70 Programs;
#L patchnumbers 0 50 1;
#L rows 25;
#L columns 2;
#L patchnames 3 15;
#L aspatches;
#L reqbank 0xF0 0x06 0 / chan 0x30 / 0x61 / patchnum / 0xF7;
#L reqpatch 0xF0 0x06 0 / chan 0x30 / 0x60 50 0xF7;
#L inbank 0xF0 0x06 0 / chan 0x10 / patchnum / 0x02 0x4E / data 334 /
   checksum 1 / 0xF7;
#L inpatch 0xF0 0x06 0 / chan 0 / 0x50 0x02 0x4E / data 334 /
   checksum 1 / 0xF7;
#L outbank 0xF0 0x06 0 / chan 0x10 / patchnum / 0x02 0x4E / data 334 /
   checksum 1 / 0xF7;
#L outpatch 0xF0 0x06 0 / chan 0 / 0x50 0x02 0x4E / data 334 /
   checksum 1 / 0xF7;
#L param 0xF0 / is $i2 / is $i4 / 0xF7;
#L sendonselect;
#L columnwidth 80;
#L sizepatch 167;
#L sizebank 8350;
#L editor pcmedit;
#L compress 2;
#L end;
```

$i4 The index into the compressed patch buffer of the parameter as defined by the argument to **libto**

$i5 The current MIDI channel

$i6 The current patch number being sent or received

When receiving, the expression will be evaluated and if the incoming byte matches the result of the statement, the pointer will move to the next byte in the matching process. There is a limitation of 32 is statements in a single **lib** script file. If you include a slash in an is expression statement, be sure to put a backslash before it.

mode *mode-statement-number*

mode allows you to define a script "macro" which can be inserted into a transmission statement. mode is designed for synthesizers that need to be put into a special mode with a MIDI message before they're ready to receive system exclusive data. You can have up to 8 mode statements, numbered 1 to 8. Here's an example:

```
#L mode 1 0xF0 0x42 / chan 0x30 / 0x24 0x4E 0x04 0x10 0xF7;
```

The above statement *defines* mode number 1. The following statement will *use* mode number 1. It precedes the actual sending of a patch, insuring that the device is set to the mode in which it can receive the patch.

```
#L outpatch mode 1 / 0xF0 0x42 / chan 0x30 / 0x24 / data 86 / 0xF7;
```

Modes are currently not used in reception statements.

patchnum

When sending, patchnum is set to the number of the patch currently being sent. If the first patch number starts at 1 (as defined in the patchnumbers statement), this value will have had 1 subtracted from it.

MIDI Communication Keywords

In the words defined below, *message* refers to a statement made up sub-statements as described in the preceding section.

reqbank *message*

Specifies the data that will be sent to a device to request a bank. This data will be sent when the **lib** object receives the load message with no arguments.

reqpatch *message*

Specifies the data that will be sent to a device to request a patch. The patch number (for purposes of the patchnum sub-statement) is the argument when the **llb** object receives the load message, i.e. load 5 will request patch number 5. If it's only possible to load the "active" patch (as in a DX7), the presence of the argument to load serves to distinguish loading a patch from a bank.

inbank *message*

Specifies the data to be expected when a bank is being received. Each element of the message is matched in sequence with incoming MIDI data. At the point the messages differ, the matching is abandoned and the **llb** object waits to see the first byte of the inbank message in subsequent incoming data. If the entire message is seen, the data is placed in the **llb** object's buffer and displayed in its library window.

inpatch *message*

Specifies the data to be expected when a patch is being received.

outbank *message*

Specifies the data that will be sent to dump a bank to the device.

outpatch *message*

Specifies the data that will be sent to dump a patch to the device. This message is executed in response to "send on select" so the patch should be sent to the edit buffer of the device if possible.

param *message*

Specifies the data for a parameter change message. Parameter change messages are sent in response to numbers coming into **llbto** objects or the send message sent to a **llb** object. You'll probably want to use is statements in defining a param message. Refer to *Definition of Communication Sub-Statements* above for information about is statements and the variables you can use inside them to refer to parameter numbers and values.

mode *mode-statement-number message*

Defines a mode message macro which can be referred to with the mode sub-statement. Up to eight modes, numbered 1 to 8, can be defined. Typically a mode message definition will be an entire MIDI system exclusive message which puts a synthesizer in a particular mode. The **llb** object will not send a mode message if it was the most recent mode sent.

columnwidth *column-width-in-pixels*

Specifies the width of a column in the library window. 80 pixels is a reasonable width (and is used as a default if you don't include a columnwidth statement in your script).

sendonselect

Specifies that the currently selected patch in a library window should be sent to the synthesizer.

aspatches

Specifies that the MIDI device does not have any "bank" level system exclusive dump messages. This modifies the meaning of the MIDI Communication statements discussed below. Examples of devices which operate this way are the CZ-101 and the PCM70.

editor *patcher-name*

Defines patcher *patcher-name* as a patcher which will be loaded when the user double-clicks on the number of a patch in the librarian. The editor Patcher file should be located in the same folder as the file which contains the **lib** object or in the Max search path. Refer to the descriptions of **libto**, **libfrom**, and **libswitch** for information about how to create an editor.

MIDI Communication Statements

Each of the MIDI Communication Statements specifies a particular protocol for sending and receiving data. For sending, the statements define what bytes to send. For receiving, the statements define a pattern that the **lib** object will look for to capture the data in an incoming system exclusive message. Since many **lib** objects might all be listening to the same MIDI input, it's important to be able to distinguish one device's sysex from another. This can even be done with multiple devices connected to the same input, since lib objects can distinguish the MIDI channel (or "device ID") number if necessary.

Communication Statements are made up of a keyword followed by "sub-statements" which are separated by a slash (/) surrounded by spaces. An entire statement might look like this:

```
#L outbank 0xF0 0x42 / chan 0x30 / 0x24 0x4C / data 8572 / dc / 0xF7;
```

Definitions of Communication Sub-Statements

(Numbers)

If sending or expecting literal numbers (such as the start or end of system exclusive), a string of numbers constitute a sub-statement. Hex numbers are preceded with a 0x. Example:

```
0xF0 0x42 0x24 45 0xF7
```

dc

Expect a "don't care" byte. Only useful for input, where the information is not needed and you don't want to match it. For example, often the size of the data in the bank or patch is sent immediately before the data itself. When the size is always constant (and you already know what it will be) this is irrelevant data which can be matched with a dc statement. Each dc statement matches a single MIDI byte.

chan *mask-value*

Send or expect to match the MIDI channel (as set with the ID numerical in the library window, and/or specified as an argument to the lib object). *mask-value* is a value which is or-ed into the channel value (which will range from 0 to 15 for channels 1 to 16). For example, you might see something in a device's MIDI specification that looked something like...

3cH (where c = device MIDI channel)

This usually means that the high nibble is always 3, and the lower nibble contains the MIDI channel. This could be sent or matched with:

```
chan 0x30
```

checksum *method-number*

Send a checksum of the data bytes (for incoming data, this byte will be ignored). *method-number* is either 1 for a straight sum of the data bytes (modulo 128), or 2 for a 2's compliment of the sum.

data *bytes-to-send-or-expect*

Send or capture the data bytes in a bank or patch. The size should be the uncompressed number of bytes in the patch or bank. When receiving, the bytes will be captured and subsequently compressed (if applicable). When sending, the bytes are uncompressed (if applicable) and sent at this point in the communication statement.

is *expr-statement*

The is keyword can be followed by a C-like expression which is evaluated by the Max **expr** object. Several of the "argument" variables (refer to the description of **expr**) have been defined for use in **lib** script files. The first four only apply to parameter change messages. They are:

$i1 The "parameter number" as defined by the param argument to **libto** (explained in the description of the **libto** object)

$i2 The lower eight bits of the parameter value from **libto**

$i3 The upper eight bits of the parameter value from **libto**

end

Ends the definition of a librarian.

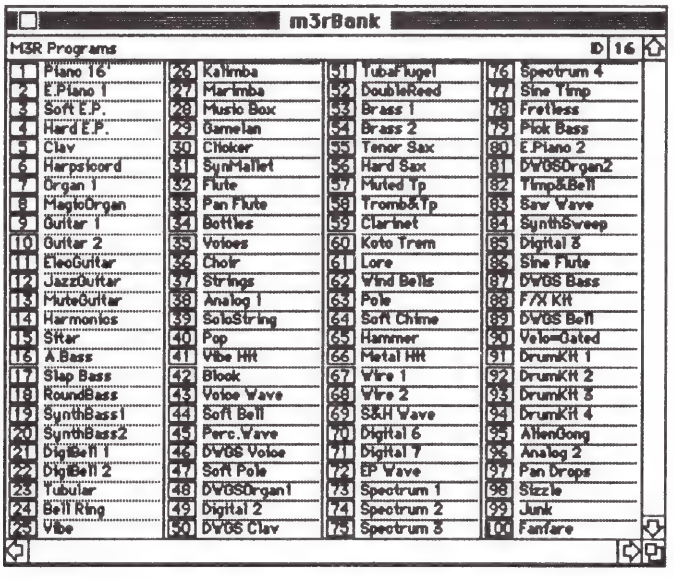
patchnumbers *starting-number number-of-patches numbering-system*

Specifies how patches are numbered, starting at *starting-number* (either 0 or 1). *number-of-patches* is the total number of patches in the group that the lib will be receiving. *numbering-system* is 1 for decimal, 2 for 11-88 "LED" format, and 3 for hexadecimal. For best results with numbering system 2, the starting patch number should be 0.

rows *number-of-rows*

columns *number-of-columns*

Specifies the number of displayed rows and columns in the library window. *number-of-rows* * *number-of-columns* should be equal to the number of patches above. For example, a window with 25 rows and 4 columns (100 total patches) would look like this:



compress *method*

Specifies how the data is transformed from a series of 7-bit numbers in the system exclusive dump message into its internal format in memory (and vice versa). Often the compressed size will be a different number of bytes (either larger or smaller). "Internal" compression methods are specified by number. Other compression methods can be specified by name (for example `#L compress KorgMeth`) and written as C code resources. If no compression is needed, you can skip the compress statement.

Internal Method Numbers:

1 7-into-8, 8-into-7, where the high bits of the subsequent 7 numbers are placed in the first 7-bit number (of 8).

2 8 bits sent as two 4-bit nibbles, lowest 4 bits first

3 8 bits sent as two 4-bit nibbles, highest 4 bits first

4 Not implemented

5 Converts DX7 128-byte format into 155-byte format

For C programmers, remember that you can invent any format you want to compress the data by writing a code resource. An example code resource, which performs the same compression as internal method number 2, is included in the development materials supplied with Max.

Note that many synthesizers don't need compression methods. Most Roland equipment is designed this way.

sizebank *size-of-bank-in-bytes*

Specifies the size of a compressed bank in bytes.

sizepatch *size-of-patch-in-bytes*

Specifies the size of a compressed patch in bytes.

patchnames *start-index end-index*

Specifies the starting and ending index of the patch name that is included within a patch. For example, if the patch name consisted of the first 10 characters, you would use the statement...

```
#L patchnames 0 9;
```

Note that these indices are in the compressed patch, not the data bytes of a patch as they are received. If patch names are not included within the patch, use the excluded statement instead of patchnames.

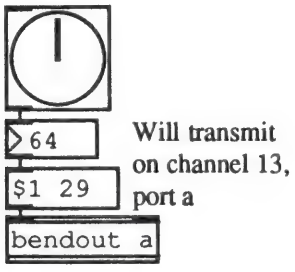
excluded *length-of-name*

Specifies that the patch type doesn't contain names, and you would like names "tagged" to each patch when stored in files and displayed on-screen. *length-of-name* specifies the number of bytes to be given to each name. A length between 10 and 18 characters is reasonable.

Transmit MIDI pitch bend messages

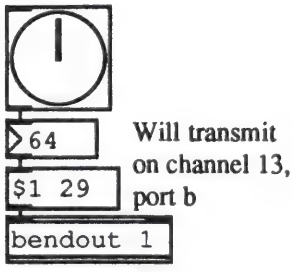
bendout

Examples



Will transmit on channel 13, port a

Letter argument transmits to only one port



Will transmit on channel 13, port b

Otherwise, number specifies both port and channel

See Also

bendin	Ouput incoming MIDI pitch bend values
midiout	Transmit raw MIDI data
xbendout	Prepare extra precision MIDI pitch bend messages
xbendin	Interpret extra precision MIDI pitch bend values
Ports	How ports are specified
Tutorial 16	More MIDI ins and outs

Borax

Report current information about note-ons and note-offs

Input

int — In left inlet: The number is the pitch value of a MIDI note-on message or note-off message (note-on with a velocity of 0). The pitch is paired with the velocity in the middle inlet. **Borax** ignores note-on messages for pitches it is already holding, and ignores note-off messages for pitches that have already been turned off. If the note is not a duplicate, **Borax** sends out the pitch and velocity values, as well as other information.

In middle inlet: The number is stored as the velocity, to be paired with pitch numbers received in the left inlet.

float — In middle inlet: Converted to int.

list — In left inlet: The second number is stored as the velocity, and the first number is used as the pitch, of a pitch-velocity pair. If the note is not a duplicate, **Borax** sends out the pitch and velocity values, as well as other information.

delta — In left inlet: Causes the *delta time* (the time elapsed since the last note-on) and the *delta count* (the number of delta times that have been reported) to be sent out.

bang — In right inlet: Resets **Borax** by sending note-offs for all notes currently being held, erasing **Borax**'s memory of all notes received, and setting its counters and its clock to 0.

Arguments

None.

Output

int — Out left outlet: Each note-on received by **Borax** is assigned a unique number, equal to the total count of note-ons received (since the last reset). That number is sent out when the note-on is received, and the same number is sent out when the note is turned off.

Out 2nd outlet: Each note is also assigned a unique voice number, equal to the lowest available number. (A voice becomes available when the note assigned to it is turned off.) That number is sent out when the note-on is received, and the same number is sent out when the note is turned off.

Out 3rd outlet: The number of notes being held by **Borax** is sent out each time a note-on or a note-off is received.

Out 4th outlet: The pitch of the note-on or note-off is sent out.

Out 5th outlet: The velocity of the note-on or note-off is sent out.

Store numbers to view or edit

capture

Input

int — Numbers are stored in the order in which they are received.

list — All ints in the list are stored in order from first to last. Floats and symbols are ignored.

clear — Erases the contents of a **capture** object.

Arguments

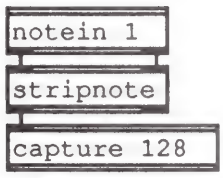
int — Optional. The first argument sets a maximum number of ints to store. If there is no argument, **capture** will store up to 512 numbers. Once the maximum has been exceeded, the earliest stored number is dropped as each new number is received.

x or m — Optional. If the second argument is x, all numbers will be displayed in hexadecimal form in the editing window. If the second argument is m, numbers less than 128 are displayed in decimal, and numbers greater than 128 are in hexadecimal. If there is no argument, all numbers are displayed in decimal.

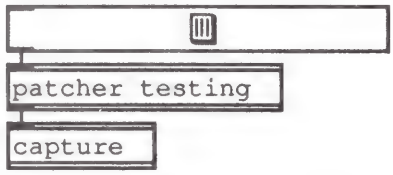
Output

There are no outlets. Double-clicking on **capture** (when the Patcher window is locked) opens an editing window in which the stored numbers can be viewed and edited. Editing the window does not actually alter the contents of capture, but is useful for cutting and pasting values into a **table** or a separate file. (Although **capture** can continue to store numbers while the editing window is open, the editing window is not updated. It must be closed and reopened to view the newly stored numbers.)

Examples



Collect numbers to paste into a table...



...or just to see what's been going on

See Also

Debugging	Techniques for debugging patches
Text	Format numbers as a text file
Tutorial 34	Managing raw MIDI data

cd

Control CD Audio with Apple CD ROM driver

Input

int — 0 pauses the CD drive, stopping the sound immediately. A number between 1 and the number of tracks begins playing the CD at the beginning of the numbered track.

list — A list of three numbers specifies an absolute minute, second, and block (1/75th of a second). The CD drive seeks to the specified location and begins playing.

search — The word search, followed by a list of three numbers, specifies an absolute minute, second, and block. The CD drive seeks to the location but does not begin playing. Send the list of three numbers (without the word search) to begin playing at this location. Using search seems to make only a slight improvement in the time it takes to get the CD to respond after sending it a command.

stop — With no arguments, stop pauses the CD drive. Followed by a list of three numbers, stop sets an automatic stop point at a specified minute, second, and block. However, this "feature" of the CD driver seems rather useless, since there is no way to play beyond the set "stop point" ever again, until you restart the computer!

Arguments

int — The **cd** object takes two arguments, both are optional (although to specify the second argument, you must type both arguments). The first argument is an identifier, for use when multiple CD drives are present. If only one drive is being used, any number can be used for the identifier. The first drive is 0, the second 1, etc. (You have *three* CD drives? Lucky fool!) The second argument specifies how often the **cd** object will report the time location of the drive when it is playing. A second argument of 0 means that the cd object will never report where the CD drive is playing.

Output

When the **cd** object is created, it prints the current CD's *table of contents* (the minute, second, and block locations of the start of each track) in the Max window. You can use this information as a guide when you specify CD locations in minute/second/block format.

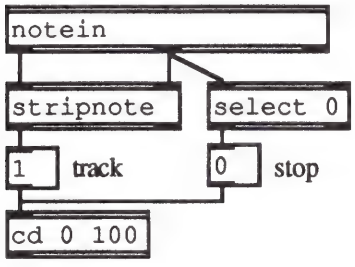
Also, as noted above, if the second argument to the **cd** object is 0 (the default), no time location output is generated. Otherwise, the numbers are sent out at an interval specified by the second argument.

int — Out left outlet: The absolute minute (starting at 1) of the CD drive when it is playing.

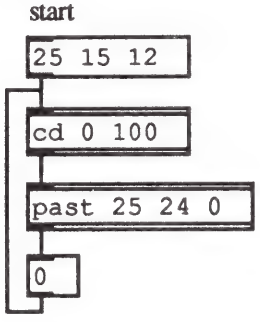
Out middle outlet: The absolute second (starting at 0) of the CD drive when it is playing.

Out right outlet: The absolute block number (starting at 0) of the CD drive when it is playing. A second of sound on a CD is divided into 75 blocks. Each block represents a 75th of a second, or 13 and 1/3 milliseconds.

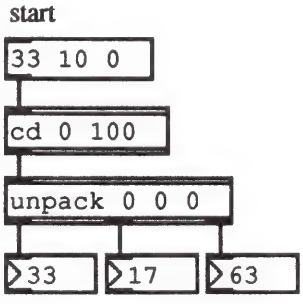
Examples



Begin playing track 1 on note-on, stop (pause CD drive) on note-off



Clicking on message box starts playing for 8 seconds and 63 blocks before turning off (triggered by the output of **past***)*



The Number boxes display the current time location of the CD drive

See Also

past Report when input increases beyond a certain value

change

Filter out repetitions of a number

Input

int — The number is sent out the outlet only if it is different from the currently stored value. Replaces the stored value.

float — Converted to int.

set — The word set, followed by a number, replaces the stored value without triggering output.

Arguments

int — Optional. Initial value for comparison to incoming numbers. If there is no argument, the initial value is 0.

Output

int — Out left outlet: The number received in the inlet is sent out only if it is different from the stored value.

Out middle outlet: If the stored value is 0 and the input is not 0, 1 is sent out; otherwise nothing is sent out.

Out right outlet: If the stored value is not 0 and the input is 0, 1 is sent out; otherwise nothing is sent out.

Examples

Filter out undesirable repetitions

See Also

TogEdge	Report a change in zero/non-zero values
!=	Compare two numbers, output 1 if not equal
Tutorial 15	Making decisions with comparisons

Send bang when Patcher window is closed

closebang

Input

There are no inlets. Output occurs when the Patcher window is closed.

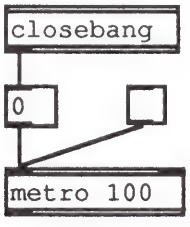
Arguments

None.

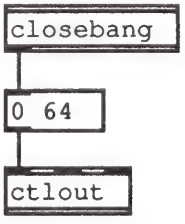
Output

bang Sent automatically when the Patcher window is closed.

Examples



Stop a process when window is about to be closed.



...or turn off sustain pedal

See Also

active	Send 1 when window is active, 0 when window is inactive
button	Flash on any message, send a bang
loadbang	Send a bang automatically when patch is loaded
Tutorial 40	Automatic actions

coll

Store and edit a collection of different messages

Input

list — The first number is used as the *address* (the storage location within **coll**) at which to store the remaining items in the list. The address will always be stored as an int.

int — The number refers to the address of a message stored in **coll**. If a message is stored at that address, the message is sent out the left outlet.

store — The word store, followed by some symbol (usually a word), followed by a message, stores the message at an address named by the symbol. (Example: store triad 0 4 7 will store the list 0 4 7 at an address named triad.)

symbol — The symbol refers to the address of a message stored in **coll**. If a message is stored at the address named by the symbol, the message is sent out the left outlet. The symbol may, but need not necessarily, be preceded by the word symbol.

float — Converted to int when received alone as an address. Left alone when part of a stored message.

goto — The word goto, followed by a number or a symbol, sets a pointer at the address specified by the number or symbol. If no such address exists, the pointer is set at the beginning of the collection. The pointer is set at the beginning of the collection initially, by default.

next — Sends the address pointed to by the pointer out the right outlet, and sends the message stored at that address out the left outlet, then sets the pointer to the next address. If the address is a symbol rather than a number, 0 is sent out the right outlet. If the pointer is currently at the last address in the collection, it *wraps around* to the first address. (Note: Number addresses are stored in ascending order. Symbol addresses are stored in the order in which they were added to the collection, after all of the number addresses.)

bang — Same effect as next.

prev — Causes the same output as the word next, but the pointer is then decremented rather than incremented. If the pointer is currently at the first address in the collection, it *wraps around* to the last address.

assoc — The word assoc, followed by a symbol and a number, *associates* the symbol with the address specified by the number, provided that the number address already exists. From then on, any reference to that symbol will be interpreted by **coll** as a reference to the number address. Each number address can have only one symbol associated with it. (Note: If the symbol was already being used as an address, or was already associated with a number address, the message that was stored at that address is removed.)

deassoc — The word deassoc, followed by a symbol and a number, removes the association between the symbol and the number address. The symbol no longer has any meaning to **coll**.

Store and edit a collection of different messages

coll

dump — Sends *all* of the stored addresses out the right outlet and *all* of the stored messages out the left outlet, in the order in which they are stored.

clear — Erases everything from the collection.

remove — The word remove, followed by a number or a symbol, removes that address and its contents from the collection.

delete — Functions similarly to the word remove, except that if the specified address is a number, all addresses of a greater number are decremented by 1.

insert — The word insert, followed by a number and a message, inserts the message at the address specified by the number, incrementing all equal or greater addresses by 1 if necessary.

length — Counts the number of messages contained in **coll** and sends the number out the left outlet.

merge — The word merge, followed by an address and a message, appends its message at the end of the message already stored at that address. If the address does not yet exist, it is created.

nth — The word nth, followed by an address and a number, gets the *nth* item (specified by the number) from the message at that address, and sends it out the left outlet. (Example: nth pgms 4 outputs the 4th item in the message stored at the address named pgms.)

sub — The word sub, followed by an address, an item number, and another number, replaces one item stored at the address. (Example: sub pgms 4 7 puts the number 7 in place of the 4th item of the message stored at the address pgms.) Only number values—not symbols—can be substituted in this manner.

flags — Normally, the contents of **coll** are not saved as part of the patch when the Patcher window is closed. The message flags 1 0 sets the **coll** object to save its contents as part of the Patcher that contains it. The message flags 0 0 causes the contents of the **coll** not to be saved with the Patcher that contains it.

(Get Info...) — A **coll** object can be set to save its contents as part of the patch that contains it. When the Patcher window is unlocked, select the **coll** object, choose **Get Info...** from the Max menu, and check *Save coll with Patcher*.

write — Calls up the standard **Save As...** dialog box, enabling the user to save the contents of **coll** as a separate file.

read — Calls up the standard **Open...** dialog box, enabling the user to load the contents of another file into **coll**.

coll

Store and edit a collection of different messages

refer — The word refer, followed by the name of another **coll** object, changes the **coll** receiving the message to refer to the data in the named **coll** object.

In addition to reading messages in from another file and storing messages via the inlet, one can also enter messages in **coll** by typing. Double-clicking with the mouse on the **coll** object displays the contents as text in an editing window which the user can modify.

In order to edit a collection by hand or read in from another file, it is essential to know the correct text format for the contents of a **coll** object. Each message is stored in the **coll** object on a separate line. The format of each line is as follows: the address (an int or a symbol), any symbols associated with that address (if the address is an int), a comma (to separate the address from the data it contains), the data (anything), and a semicolon to indicate the end of each line. In a line such as...

```
3 reset, set 4.7;
```

...3 is the number of the address, reset is a symbol associated with that address, and the message it contains is set 4.7. Here is how we would store the numbers 100, 200, 300, and 400 with the addresses 1, 2, 3, and 4.

```
1, 100;
2, 200;
3, 300;
4, 400;
```

Arguments

any symbol — Optional. Name of a file to be read into **coll** automatically when the patch is loaded. The information in the file must be in the correct format in order to be read in by **coll**. All **coll** objects which share the same name always share the same contents. You can use the file name as an identifier for the purpose of sharing data between multiple **coll** objects, without there needing to be an actual file with the specified name.

Output

anything — Messages stored in **coll** are sent out the left outlet. If the message consists of only a single symbol, it will be preceded by the word symbol when it is sent out.

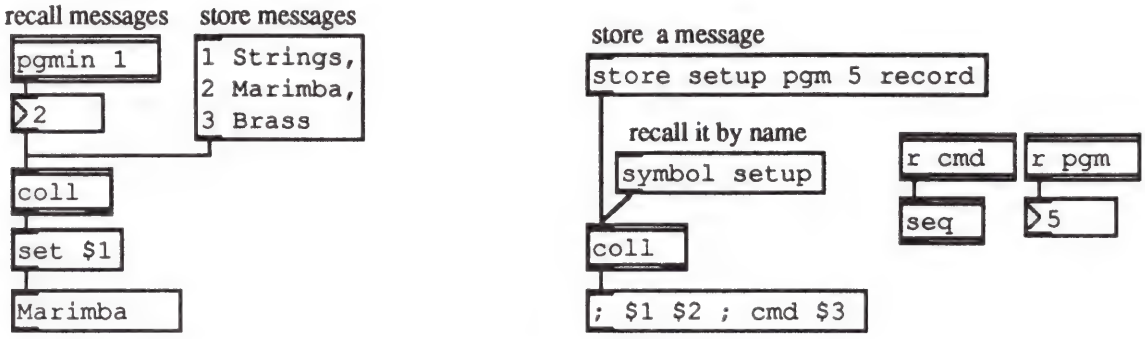
int — Out left outlet: The number of messages contained in **coll** is sent out in response to the length message.

int or symbol — Out right outlet: The address is sent out whenever a message out the left outlet is triggered by bang, dump, next, or prev.

Store and edit a collection of different messages

coll

Examples



Complex messages can be recalled with a single number or word

See Also

Data Structures	Ways of storing data in Max
bag	Store a collection of numbers
table	Store and graphically edit an array of numbers
funbuff	Store x,y pairs of numbers together
Tutorial 37	Data structures

comment

Put explanatory notes or labels in a patch

Input

anything The **comment** object has no inlets and receives no input.

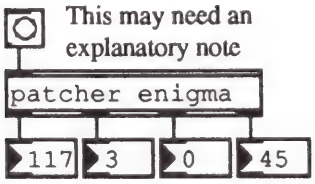
Arguments

Text is typed directly into the **comment** box when the Patcher window is in Edit mode. When the Patcher window is locked, the outline of the **comment** box disappears, and only the text is shown. The appearance of a **comment** can be modified by changing the font and by resizing its box.

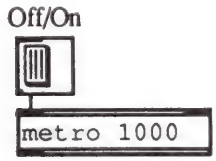
Output

A **comment** has no outlets, sends no output, and does not affect the functioning of the patch.

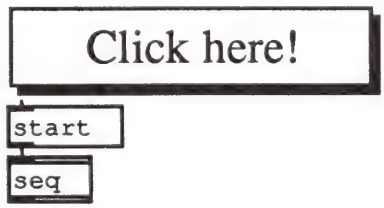
Examples



Elucidate



Label



Make functional (with **ubutton***)*

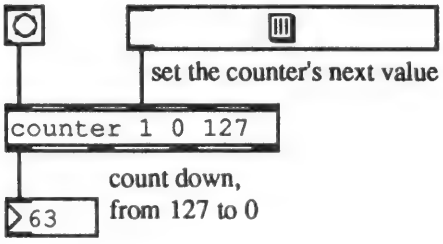
See Also

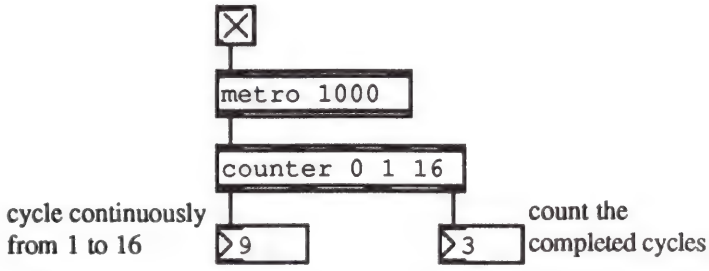
ubutton	Transparent button
Tutorial 5	toggle and comment

Count the bangs received, output the count

counter

Examples





Keep track of how many events have occurred, or create a continuous loop

See Also

Loops	Using loops to perform repeated operations
tempo	Output numbers at a metronomic tempo
Tutorial 31	Using timers

ctlin

Output incoming MIDI control values

Input

(MIDI) There are no inlets. The input is a MIDI control message received directly from the serial ports, or from the *virtual ports* of the MIDI Manager.

Arguments

a-f Optional. Specifies a single port from which to receive incoming control messages. If there is no letter present as an argument, **ctlin** can receive from all ports.

If the MIDI Manager is being used, a letter a-f specifies a virtual port. If the MIDI Manager is not being used, the letter a specifies the modem port and b specifies the printer port.

int Following the (optional) port argument, the next argument is a single controller number to be recognized by **ctlin**. If there is no controller number, or if the argument is a negative number, **ctlin** recognizes *all* controller numbers. If a single controller number *is* specified in the argument, the outlet which normally sends the controller number is unnecessary, and is not created.

Following the controller number argument is a single channel number on which to receive control messages. If the channel argument is not present, **ctlin** receives control messages on all channels. In order for this argument to be used, a controller number argument *must* precede it. To specify a channel number without specifying a controller number, use -1 for the controller number.

If a single channel number is specified as an argument, the outlet which normally sends the channel number is unnecessary, and is not created. If a port has been specified with a letter argument, channel numbers greater than 16 will be *wrapped around* to stay within the 1-16 range. If no port argument is present, a channel number can be used in place of a letter and number combination. Numbers 1-16 specify a channel number on port a, numbers 17-32 specify channels 1-16 on port b, etc.

Output

int Out left outlet: The number is the control value of an incoming MIDI control change message.

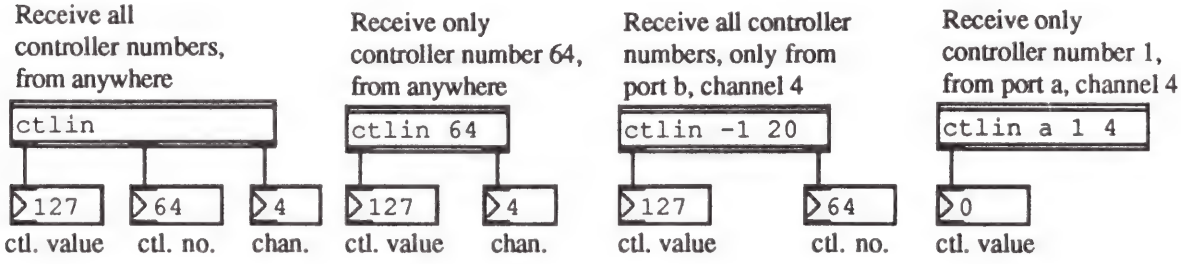
If a specific controller number is not specified as an argument, the controller number is sent out the 2nd outlet.

If a specific channel number is not included in the argument, the channel number is sent out an additional, right, outlet.

Output incoming MIDI control values

ctlin

Examples



Control messages can be filtered in a variety of ways

See Also

ctlout	Transmit MIDI control messages
MIDI	MIDI software protocol
midiin	Output incoming raw MIDI data
Ports	How MIDI ports are specified
Tutorial 16	More MIDI ins and outs

ctlout

Transmit MIDI control messages

Input

int — In left inlet: The number is used as the control value, and **ctlout** transmits a MIDI control change message. Numbers are limited between 0 and 127.

In middle inlet: The number is stored as the controller number of the control change messages transmitted by **ctlout**. Numbers are limited between 0 and 127.

In right inlet: The number is stored as the channel number on which to transmit the control messages.

float — Converted to int.

list — In left inlet: The first number is the control value, the second the controller number, and the third the channel number. **ctlout** transmits a MIDI control change message using these values.

port — In left inlet: The word port, followed by a letter a-f, specifies the port used to transmit the MIDI messages.

Arguments

a-f — Optional. Specifies the port for transmitting MIDI control messages. If there is no argument, **ctlout** initially transmits out port a, on channel 1.

If the MIDI Manager is being used, a letter a-f specifies a virtual port. If the MIDI Manager is not being used, the letter a specifies the modem port and b specifies the printer port. When a port is specified by a letter argument, channel numbers greater than 16 received in the right inlet will be *wrapped around* to stay within the 1-16 range.

int — Following the (optional) port argument, the next argument is an initial value for the controller number to be used in control messages transmitted by **ctlout**. Controller numbers are automatically limited between 0 and 127. If there is no controller number specified, the initial controller number is 1.

Following the controller number argument is an initial value for the channel number on which to transmit control messages. If the channel argument is not present, **ctlout** initially transmits control messages on channel 1. In order for this argument to be used, a controller number argument *must* precede it.

If a port has been specified with a letter argument, channel numbers greater than 16 will be *wrapped around* to stay within the 1-16 range. If no port argument is present, the channel number specifies both the port and the channel. Numbers 1-16 specify a channel number on port a, numbers 17-32 specify channels 1-16 on port b, etc.

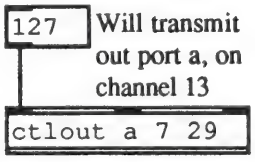
Transmit MIDI control messages

ctlout

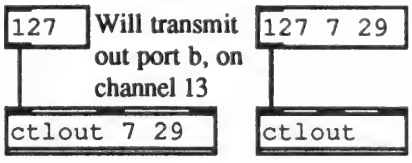
Output

(MIDI) There are no outlets. The output is a MIDI control message transmitted directly to the serial ports, or to the *virtual ports* of the MIDI Manager.

Examples



Letter argument transmits to only one port



Otherwise, number specifies both port and channel

See Also

ctlin	Ouput incoming MIDI control values
MIDI	MIDI overview and specification
midiout	Transmit raw MIDI data
Ports	How MIDI ports are specified
Tutorial 16	More MIDI ins and outs

Decode

Send 1 or 0 out a specific outlet

Input

int **Decode** acts as a hierarchical switchboard. The right inlet is the master switch, which can turn off (send 0 out) all outlets. The middle inlet is a submaster switch, which can turn on (send 1 out) all outlets, *provided* they have not all been turned off by the master switch. The left inlet can turn on one of the outlets exclusively, provided neither the submaster switch nor the master switch is active.

In left inlet: The number specifies an outlet out to turn on, turning off all other outlets. (Whenever an outlet is turned on that was previously turned off, a 1 is sent out. Conversely, whenever an enabled outlet is disabled, a 0 is sent out.) The outlets are referred to by number, beginning with 0 on the left, and numbers received in the left inlet are automatically limited between 0 and the number of outlets minus 1.

In middle inlet: Any number other than 0 enables all disabled outlets (sends a 1 out them), unless *all* outlets are disabled. When 0 is received, **Decode** turns off all outlets except the one that had previously been on.

In right inlet: Any number other than 0 disables all enabled outlets (sends a 0 out them). Once all outlets have been disabled in this manner, no outlet can be enabled until a 0 is received in the right inlet. When a 0 is received, **Decode** re-enables all outlets that it had just disabled.

float Converted to int.

Arguments

int Optional. Sets the number of outlets. The number of outlets is limited between 1 and 8. The default is one outlet.

float Converted to int.

Output

int When an outlet is enabled that was previously disabled, a 1 is sent out that outlet. When an outlet is disabled that was previously enabled, a 0 is sent out that outlet. The left outlet is initially enabled.

Reverse the sequential order of two decimal numbers

fswap

Input

float — In left inlet: The number is sent out the right outlet, then the number in the right inlet is sent out the left outlet.

In right inlet: The number is stored to be sent out the left outlet when a number is received in the left inlet.

int — If there is a float argument, the numbers are converted to float. If there is an int argument or no argument, the number received in the right inlet is stored as an int.

list — In left inlet: The numbers are stored in **fswap**. The first number is sent out the right outlet, then the second number is sent out the left outlet.

bang — In left inlet: Swaps and sends out the numbers currently stored in **fswap**.

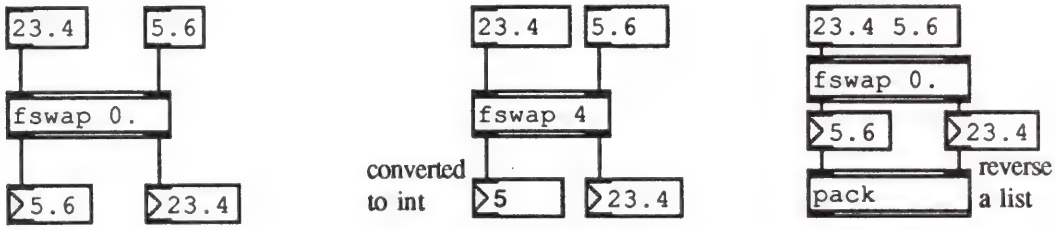
Arguments

int or float — Optional. Sets an initial value for the number which is to be sent out the left outlet. If there is no argument, the initial value is 0. If there is an int argument or no argument, an int is sent out the left outlet. (The number sent out the right outlet is *always* a float.)

Output

int — When a number is received in the left inlet, the number in each inlet is sent out the opposite outlet.

Examples



Numbers are sent out in reverse order from the order in which they were received

See Also

pack — Combine numbers together into a list
swap — Reverse the sequential order of two numbers
unpack — Break a list up into individual numbers

funbuff

Store x,y pairs of numbers together

Input

list — In left inlet: The first number is the x value, and the second number is the y value, of an x,y pair stored in **funbuff**. If the x value is the same as an x value already stored in **funbuff**, the previously stored pair is replaced by the new pair.

int — In left inlet: The number is the x value of an x,y pair. If a y value has been received in the right inlet, the two numbers are stored together in **funbuff**. Otherwise, the x value causes the corresponding y value stored in **funbuff** to be sent out the left outlet.

If there is no stored x value which matches the number received, **funbuff** uses the closest x value which is *less than* the number received, and sends out the corresponding y value.

In right inlet: The number is a y value which will be paired with the next x value received in the left inlet, and stored in **funbuff**.

clear — Erases the contents of funbuff.

goto — The word goto, followed by a number, sets a pointer to the x value specified by the number.

next — Finds the x value pointed to by the pointer (or, if the pointer points to a number not yet stored as an x value, to the next *greater* x value), and sends the corresponding y value out the left outlet. Also, **funbuff** calculates the difference between that x value and the value previously pointed to by the pointer, sends the difference out the middle outlet, and resets the goto pointer to the next greater x value.

write — Calls up the standard **Save As...** dialog box, so that the contents of **funbuff** can be saved as a separate file.

read — Calls up the standard **Open...** dialog box so that a file of x,y values can be read into **funbuff**.

Arguments

any symbol — Optional. The argument specifies the name of a file to be read into **funbuff** when the patch is loaded. Changes to the contents of one **funbuff** will not affect the contents of another **funbuff** object with the same name.

A file for **funbuff** can also be created using a text editor window, beginning the text with the word funbuff, followed by a list of space-separated numbers that specify alternating x and y values. A **funbuff** which has been saved as a file can be viewed and edited as text by choosing **Open as Text...** from the File menu. Numbers in the form of text can be pasted in from other sources such as the editing window of a **capture** object, or even from another program such as a word processor.

Store x,y pairs of numbers together

funbuff

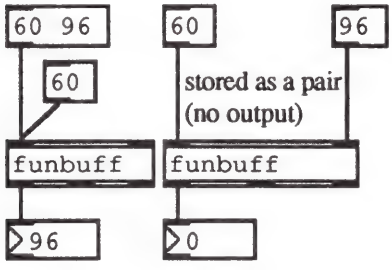
Output

int Out left outlet: When an x value is received in the left inlet, the corresponding y value is sent out. (Or, if there is no such x value yet stored in **funbuff**, the y value corresponding to the next *lesser* x value is sent out.) When the word next is received in the left inlet, **funbuff** sends out the y value that corresponds to the x value pointed to by its pointer (or, if there is no such x value, the y value of the next *greater* x value).

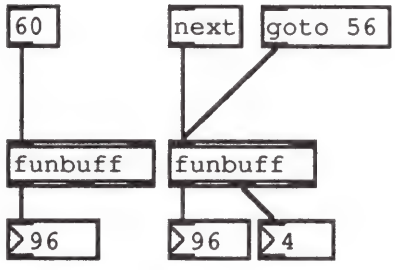
Out middle outlet: When the word next is received in its left inlet, **funbuff** sends out the difference between the x value pointed to by its pointer, and the x value *previously* pointed to, then resets the pointer to the next x value.

bang Out right outlet: When the pointer reaches the end of a **funbuff**, no numbers are sent out in response to a next message, but a bang is sent out to notify that the end has been reached.

Examples



Lists are stored as x,y pairs



An x value alone, or next, *sends out a y value*

See Also

table	Store and graphically edit an array of numbers
Tutorial 27	Your object

gate

Pass the input out a specific outlet

Input

int — In left inlet: The number specifies an open outlet through which to pass all messages received in the right inlet. A number in the left inlet does not trigger any output itself.

float — In left inlet: Converted to int.

bang — In left inlet: Reports the current open outlet, or 0 if closed, out the left outlet. This message is designed to be used in conjunction with the **grab** object.

anything — In right inlet: All messages are passed out the open outlet, specified by the number in the left inlet.

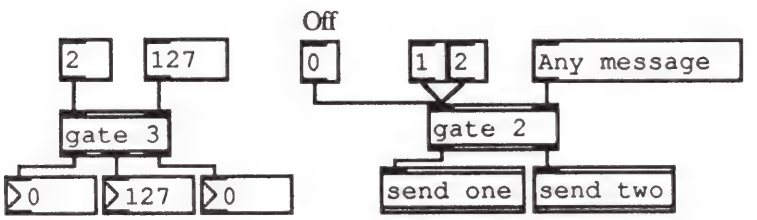
Arguments

int — Optional. Specifies the number of outlets. Limited between 1 and 10. If there is no argument, there is only one outlet.

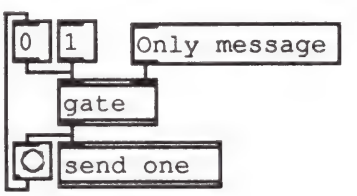
Output

anything — Messages received in the right inlet are passed out the outlet specified by the number in the left inlet. If the number in the left inlet is 0, or if no outlet number has been received yet, all messages are ignored. If the number in the left inlet is less than 0, messages are sent out the *leftmost* outlet. If it is greater than the number of existing outlets, messages are sent out the *rightmost* outlet.

Examples



Message is passed out the specified outlet



This one closes the door behind itself

See Also

Ggate — Pass the input out one of two outlets
switch — Receive the input in a specific inlet
Tutorial 17 — Gates and switches

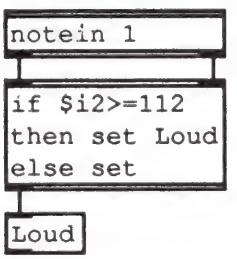
Conditional statement in if/then/else form

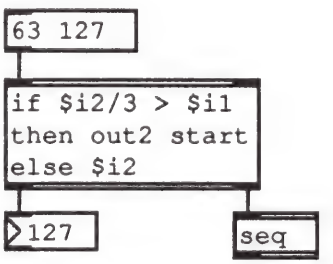
if

Output

anything — Messages are sent out the outlet. If the word out2 is present as an argument, there will be two outlets, and messages following out2 will be sent out the right outlet. If the word send is present as an argument, the word that follows it is the name of a **receive** object, and the message that follows it will be sent to the **receive** object.

Examples





Complex comparisons and results can be described in a single object

See Also

expr — Evaluate a mathematical expression
Tutorial 38 — **expr** and **if**

inlet

Receive messages from outside a Patcher

Input

anything — Each **inlet** object in a patch will show up as an inlet at the top of an object box when the patch is used inside another patch (as an object or a subpatch). Messages sent into such an inlet will be received by the **inlet** object in the subpatch.

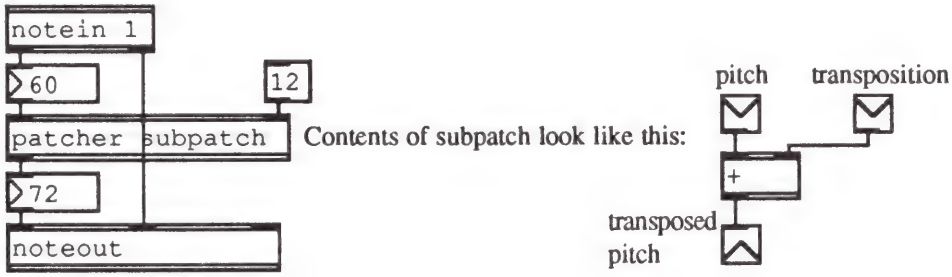
Arguments

A descriptive **Assistance** message can be assigned to an inlet by selecting the **inlet** object (when the Patcher window is unlocked) and choosing **Get Info...** from the Max menu. The **Assistance** message will appear when the patch is used as an object inside another patch.

Output

anything — In a subpatch **inlet** sends out whatever messages it receives through patch cords from the patch that contains it.

Examples



Inlets of the subpatch... *correspond to the* **inlet** *objects in the subpatch*

See Also

outlet	Send messages out from a Patcher
receive	Receive messages from other objects without patch cords
send	Send messages to other objects without patch cords
Tutorial 26	The **patcher** object

Script-driven
patch librarian

lib

Input

int — In left inlet: The numbers are individual bytes of a MIDI stream from **sysexin** or **midiin**. The **lib** object will respond to incoming MIDI system exclusive messages by storing bulk dumps from a MIDI device if the bytes match the patterns specified in the **lib** script.

int — In right inlet: Sets the **lib**'s MIDI channel. This number is displayed in the library window and can also be used by incoming or outgoing messages in the **lib** script.

send — In left inlet: send, followed by a number specifying an index into the current patch for the location of a parameter, the optional word param (or p) and a number specifying a parameter number, and a number specifying a data value for the parameter, sends a parameter change message according to the format of the param message in the lib script.

put — In left inlet: put has the format and result as the send message but will also store the data value in the current patch.

read — In left inlet: read opens a standard **Open...** dialog box to read in a file of patches into the **lib** object. Only files whose file type matches that specified in the **lib** script will be listed in the dialog box.

write — In left inlet: write opens a standard **Save As...** dialog box to save the current contents of the library in a file. The file will have the filetype specified in the **lib** script and can be opened by sending the read message to **lib**.

open — In left inlet: open opens the **lib** object's window for displaying and editing patch names. This window can also be opened by double-clicking on the **lib** object box.

Arguments

any symbol — Obligatory. The name of a script file which will be read in to define the behavior of the **lib** object. The script file can be located in the same directory as the Patcher file which contains the **lib** object, or in any folder in Max's search path.

int — Optional (follows the symbol). Sets the initial MIDI Channel. If no argument is present, the channel is 1 by default.

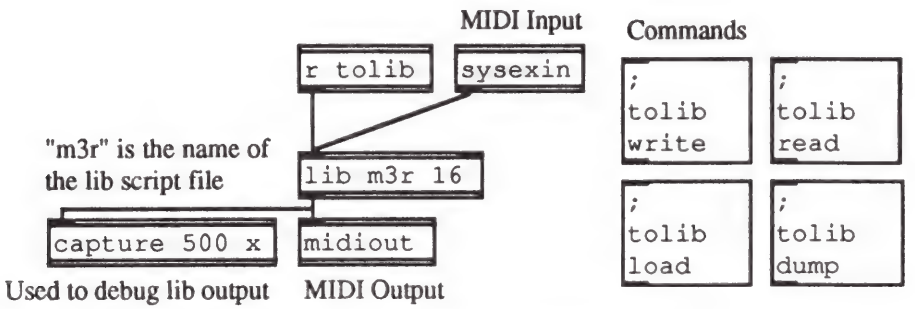
Output

int — **lib** outputs the bytes of MIDI system exclusive messages—patch dumps and parameter changes. The output of **lib** should be connected to **midiout**.

lib

Script-driven patch librarian

Examples



Typical configuation of **lib** *object*

See Also

Librarians	Creating patch editors and librarians
sysexin	Output incoming MIDI system exclusive messages
midiout	Transmit raw MIDI data
libfrom	Output stored patch value when editor is loaded
libto	Store data into an edited patch, send parameter changes
libswitch	Change the patch being edited within an editor window

Output incoming raw MIDI data

midiin

Input

(MIDI) There are no inlets. The input is a MIDI message received directly from the serial ports, or from the *virtual ports* of the MIDI Manager.

Arguments

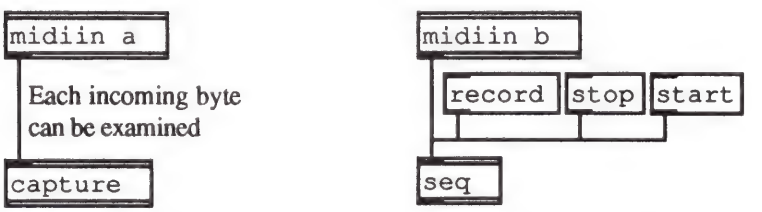
a-f Optional. Specifies the port from which to receive incoming MIDI messages. If there is no argument, **midiin** receives from port a.

If the MIDI Manager is being used, a letter a-f specifies a virtual port. If the MIDI Manager is not being used, the letter a specifies the modem port and b specifies the printer port.

Output

int All MIDI messages received from the specified port are sent out the outlet, byte-by-byte. Note that midin does not "clean up" any use of running status in the incoming MIDI stream.

Examples



MIDI messages received in a port are output by a **midiin** *object*

See Also

lib	Script-driven patch librarian
midiout	Transmit raw MIDI data
midiparse	Interpret raw MIDI data
rtin	Output incoming MIDI real time messages
sysexin	Output incoming MIDI system exclusive messages
xnotein	Interpret MIDI note messages with release velocity
xbendin	Interpret extra-precision MIDI pitch bend messages
MIDI	MIDI overview and specification
Ports	How MIDI ports are specified
Tutorial 34	Managing raw MIDI data

midiout

Transmit raw MIDI data

Input

int — The number is transmitted as a byte of a MIDI message to the specified port.

float — Converted to int.

port — The word port, followed by a letter a-f, specifies a port out which to transmit the MIDI messages.

Arguments

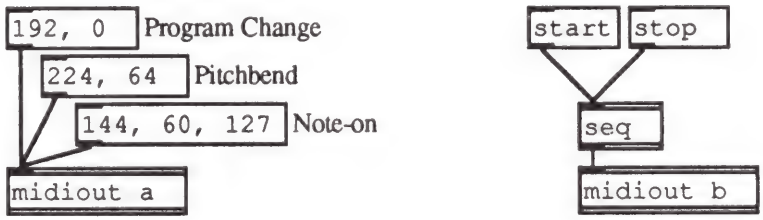
a-f — Optional. Specifies the port for transmitting MIDI data. If there is no argument, **midiout** transmits out port a.

If the MIDI Manager is being used, a letter a-f specifies a virtual port. If the MIDI Manager is not being used, the letter a specifies the modem port and b specifies the printer port.

Output

(MIDI) — There are no outlets. The output is a byte of a MIDI message transmitted directly to the serial ports, or to the *virtual ports* of the MIDI Manager.

Examples



MIDI bytes are received in the inlet and are transmitted out the specified port

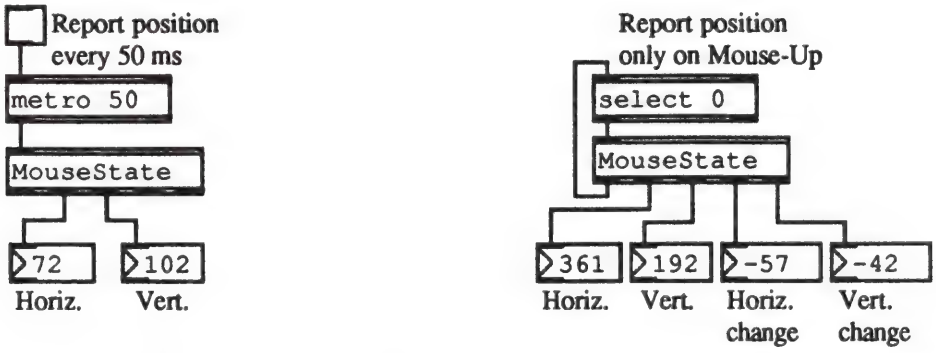
See Also

lib	Script-driven patch librarian
midiin	Ouput incoming raw MIDI data
midiformat	Prepare data in the form of a MIDI message
sxformat	Prepare MIDI system exclusive messages
MIDI	MIDI overview and specification
Tutorial 34	Managing raw MIDI data

Report the status and location of the mouse

MouseState

Examples



The mouse can provide continuous or discrete values

See Also

mousefilter	Pass numbers only when the mouse button is up
Tutorial 39	Mouse control

mtr

Multi-track sequencer

Input

record — In left inlet: Begins recording all messages received in the other inlets. The word record, followed by one or more track numbers, begins recording those tracks.

In other inlets: Begins recording messages on the track that corresponds to the inlet.

play — In left inlet: Plays back all messages recorded earlier, sending them out the corresponding outlets in the same rhythm and at the same speed they were recorded. The word play, followed by one or more track numbers, begins playing those tracks.

In other inlets: Plays back all messages on the track that corresponds to the inlet.

stop — In left inlet: Stops **mtr** when it is recording or playing. The word stop, followed by one or more track numbers, stops those tracks.

In other inlets: Stops the track that corresponds to the inlet.

next — In left inlet: Causes each track to output only the next message in its recorded sequence. When a next message is received, the track number and the delta time of each message being output are sent out the leftmost outlet as a list. The word next, followed by one or more track numbers, outputs the next message stored in that those tracks.

In other inlets: Outputs the next message stored on the track that corresponds to the inlet.

rewind — In left inlet: Resets **mtr** to the beginning of its recorded sequence. This command is used to return to the beginning of the sequence when stepping through messages with next. To return to the beginning of a sequence while playing or recording, just repeat the play or record message. When **mtr** is playing or recording, a stop message should precede a rewind message. The word rewind, followed by one or more track numbers, returns to the beginning of those tracks.

In other inlets: Returns the pointer to the beginning of the track that corresponds to the inlet.

mute — In left inlet: Causes **mtr** to stop producing output, while still continuing to "play" (still moving forward in the sequence). The word mute, followed by one or more tracks, mutes those tracks.

In other inlets: Mutes the track that corresponds to the inlet.

delay — In left inlet: The word delay, followed by a number of miliseconds, sets the first *delta time* value of each track to that number, so that all tracks begin playing back that amount of time after the play message is received.

In other inlets: Sets the initial delta time of the track that corresponds to the inlet.

Output incoming MIDI note messages

notein

Input

(MIDI) There are no inlets. The input is a MIDI note-on or note-off message received directly from the serial ports, or from the *virtual ports* of the MIDI Manager.

Arguments

a-f Optional. Specifies the port from which to receive incoming note messages. If there is no argument, **notein** receives from all channels on all ports.

a-f and int A letter and number combination (separated by a space) indicates a port and a specific MIDI channel on which to receive note messages. Channel numbers greater than 16 will be *wrapped around* to stay within the 1-16 range.

int A number alone can be used in place of a letter and number combination. Numbers 1-16 specify a channel number on port a, numbers 17-32 specify channels 1-16 on port b, etc.

If the MIDI Manager is being used, a letter a-f specifies a virtual port. If the MIDI Manager is not being used, the letter a specifies the modem port and b specifies the printer port.

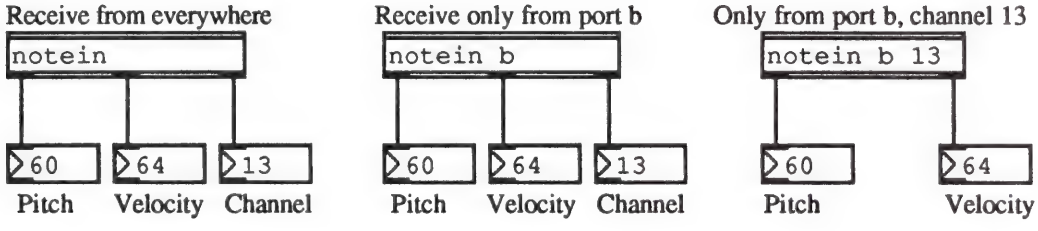
Output

int Out left outlet: The number is the pitch value of the incoming note message.

Out 2nd outlet: The number is the velocity of the incoming note-on message if non-zero, 0 for a note-off message. To receive release velocity, use **xnotein**.

If a specific channel number is included in the argument, there are only two outlets. If there is *no* channel number specified by the argument, **notein** will have a third outlet, on the right, which will output the channel number of the incoming note message.

Examples



Note-on messages can be received from everywhere, a specific port, or a specific port and channel

notein

Output incoming MIDI note messages

See Also

noteout	Transmit MIDI note messages
xnotein	Interpret MIDI note messages with release velocity
midiin	Output incoming raw MIDI data
Ports	How ports are specified
Tutorial 12	Sending and receiving MIDI notes

Transmit MIDI note messages

noteout

Input

int — In left inlet: The number is the pitch value of a MIDI note message transmitted on the specified channel and port. Numbers are limited between 0 and 127.

In middle inlet: The number is stored as the velocity of a note message, to be used with pitch values received in the left inlet. Numbers are limited between 0 and 127. 0 is considered a note-off message, 1-127 are note-on messages.

In right inlet: The number is stored as the channel number on which to transmit the note-on messages.

float — Converted to int.

list — In left inlet: The first number is used as the pitch, the second number is used as the velocity, and the third number is used as the channel, of a transmitted MIDI note message.

port — In left inlet: The word port, followed by a letter a-f, specifies the port used to transmit the MIDI messages.

Arguments

a-f — Optional. Specifies the port for transmitting MIDI note messages. Channel numbers greater than 16 received in the right inlet will be *wrapped around* to stay within the 1-16 range. If there is no argument, **noteout** initially transmits out port a, on MIDI channel 1.

a-f and int — A letter and number combination (separated by a space) indicates a port and a specific MIDI channel on which to transmit note messages. Channel numbers greater than 16 will be *wrapped around* to stay within the 1-16 range.

int — A number alone can be used in place of a letter and number combination. Numbers 1-16 specify a channel number on port a, numbers 17-32 specify channels 1-16 on port b, etc. In this case, channel numbers received in the right inlet will specify both the port and the channel number.

If the MIDI Manager is being used, a letter a-f specifies a virtual port. If the MIDI Manager is not being used, the letter a specifies the modem port and b specifies the printer port.

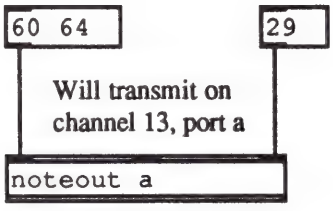
Output

(MIDI) — There are no outlets. The output is a MIDI note-on message transmitted directly to the serial ports, or to the *virtual ports* of the MIDI Manager.

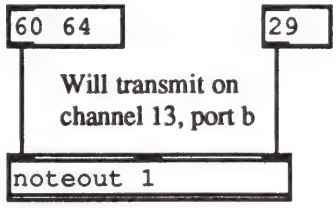
noteout

Transmit MIDI note messages

Examples



Letter argument transmits to only one port



Otherwise, number specifies both port and channel

See Also

notein	Ouput incoming MIDI note messages
xnoteout	Format MIDI note messages with release velocity
midiout	Transmit raw MIDI data
Ports	How ports are specified
Tutorial 12	Sending and receiving MIDI notes

Display and output a number

Number box

Input

int or float — The number received in the inlet is stored and displayed in the **Number box** and sent out the outlet. A float is converted to int by an int **Number box**, and vice versa.

When the Patcher window is locked, numbers can be entered into the **Number box** by clicking on it with the mouse and typing in a number on the Macintosh keyboard. Typing the Return key or the Enter key, or clicking *outside* the **Number box**, sends the number out the outlet.

Dragging up and down on the **Number box** with the mouse (when the Patcher window is locked) moves the displayed value up and down, and outputs the new values continuously. In the float **Number box**, dragging to the left of the decimal point changes the value in increments of 1. Dragging to the right of the decimal point changes the fractional part of the number in increments of 0.01.

bang — Sends the currently displayed number out the outlet.

set — The word set, followed by a number, sets the stored and displayed value to that number without triggering output.

min — The word min, followed by a number, sets the minimum value that can be displayed by the **Number box**.

max — The word max, followed by a number, sets the maximum value that can be displayed by the **Number box**.

(typing) — When a **Number box** is *highlighted* (indicated by a filled-in triangle) in a Patcher window, numerical keyboard input is sent to the **Number box** to change its value. Clicking the mouse or pressing Return or Enter stores a pending typed number.

Arguments

You can set minimum and maximum integer limits on stored and displayed values by choosing **Get Info...** from the Max menu on a selected **Number box**. Numbers which exceed the limits can still be *passed through* the **Number box** or typed on the Macintosh keyboard. When the patch is loaded, the **Number box** is initially set to the minimum value, or to 0 if there is no minimum, or the minimum is less than 0. A newly created **Number box** has no minimum or maximum value.

Other options available in the **Get Info...** dialog box are: *Draw in Bold* (to display in bold typeface), *Draw Triangle* (to have an arrow pointing to the number, giving it a distinctive appearance), *Display in Hex* (to display an int in hexadecimal form), *MIDI Note Names* (to display an int as its MIDI note equivalent), *Send on Mouse-Up* (to send a number only when the mouse button is released, rather than continuously), and *Can't Change* (to disallow changes with the mouse or the Macintosh keyboard).

Number box

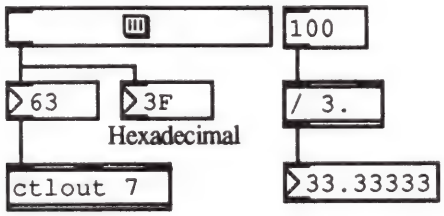
Display and output a number

When the *Display in Hex* or the *MIDI Note Names* option is checked, numbers must be typed into the **Number box** in the specified format.

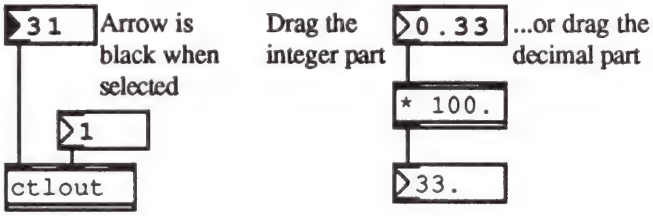
Output

int or float — The number displayed in the **Number box** is sent out the outlet. Numbers received in the inlet or typed on the Macintosh keyboard can exceed the limits of the **Number box**, but the value that is stored and displayed will automatically be limited to the specified range.

Examples



Displays numbers passing through



Can be used to output numbers

See Also

Tutorial 3	About numbers
Tutorial 10	Number boxes

If a number is greater than previous numbers, output it

Peak

Input

int — In left inlet: If the input is greater than the value currently stored in **Peak**, it is stored as the new peak value and is sent out.

In right inlet: The number is stored in **Peak** as the new peak value, and is sent out.

float — Converted to int.

list — In left inlet: The second number is stored as the new peak value and is sent out, then the first number is received in the left inlet.

Arguments

None. The initial value stored in **Peak** is 0.

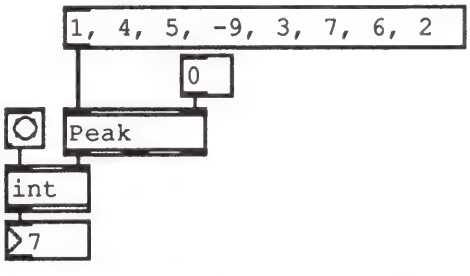
Output

int — Out left outlet: New peak values are sent out. (A number received in the right inlet is *always* the new peak value.)

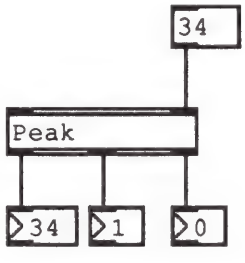
Out middle outlet: If the number received is a new peak value, the output is 1. If the number received in the left inlet is *not* a new peak value, the output is 0.

Out right outlet: If the number received is a new peak value, the output is 0. If the number received in the left inlet is *not* a new peak value, the output is 1.

Examples



Find the greatest in a series of numbers



A number in the right inlet always sets a new peak

See Also

maximum — Output the greater of two numbers
Trough — If a number is less than previous numbers, output it
> — *Is greater than,* comparison of two numbers

pgmin

Output incoming MIDI program change values

Input

(MIDI) There are no inlets. The input is a MIDI program change message received directly from the serial ports, or from the *virtual ports* of the MIDI Manager.

Arguments

a-f Optional. Specifies the port from which to receive incoming program change messages. If there is no argument, **pgmin** receives from all channels on all ports.

a-f and int A letter and number combination (separated by a space) indicates a port and a specific MIDI channel on which to receive program change messages. Channel numbers greater than 16 will be *wrapped around* to stay within the 1-16 range.

int A number alone can be used in place of a letter and number combination. Numbers 1-16 specify a channel number on port a, numbers 17-32 specify channels 1-16 on port b, etc.

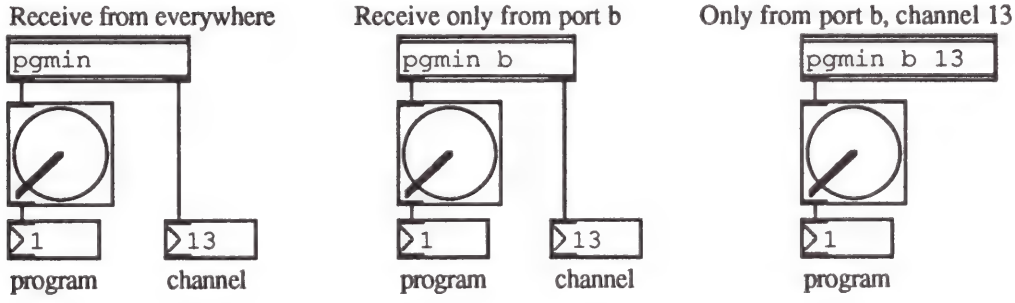
If the MIDI Manager is being used, a letter a-f specifies a virtual port. If the MIDI Manager is not being used, the letter a specifies the modem port and b specifies the printer port.

Output

int If a specific channel number is included in the argument, there is only one outlet. The output is the incoming program number on the specified channel and port.

If there is no channel number specified by the argument, **pgmin** will have a second outlet, on the right, which will output the channel number of the incoming program change message.

Examples



Program changes can be received from everywhere, a specific port, or a specific port and channel

Output incoming MIDI program change values

pgmin

See Also

pgmout	Transmit MIDI program change messages
midiin	Output incoming raw MIDI data
Ports	How ports are specified
Tutorial 16	More MIDI ins and outs

pgmout

Transmit MIDI program change messages

Input

int — In left inlet: The number is transmitted as a program change value on the specified channel and port. Numbers are limited between 0 and 127.

In right inlet: The number is stored as the channel number on which to transmit the program change messages.

float — Converted to int.

list — In left inlet: The first number is the program number, and the second number is the channel, of a MIDI program change message, transmitted on the specified channel and port.

port — In left inlet: The word port, followed by a letter a-f, specifies the port used to transmit the MIDI program change messages.

Arguments

a-f — Optional. Specifies the port for transmitting MIDI program change messages. Channel numbers greater than 16 received in the right inlet will be *wrapped around* to stay within the 1-16 range. If there is no argument, **pgmout** initially transmits out port a, on MIDI channel 1.

a-f and int — A letter and number combination (separated by a space) indicates a port and a specific MIDI channel on which to transmit program change messages. Channel numbers greater than 16 will be *wrapped around* to stay within the 1-16 range.

int — A number alone can be used in place of a letter and number combination. Numbers 1-16 specify a channel number on port a, numbers 17-32 specify channels 1-16 on port b, etc. In this case, channel numbers received in the right inlet will specify both the port and the channel number.

If the MIDI Manager is being used, a letter a-f specifies a virtual port. If the MIDI Manager is not being used, the letter a specifies the modem port and b specifies the printer port.

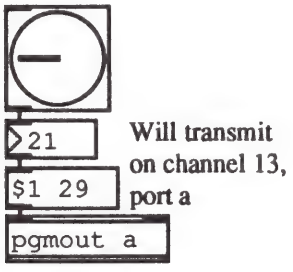
Output

(MIDI) — There are no outlets. The output is a MIDI program change message transmitted directly to the serial ports, or to the *virtual ports* of the MIDI Manager.

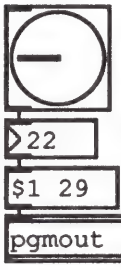
Transmit MIDI program change messages

pgmout

Examples



Will transmit on channel 13, port a



Will transmit on channel 13, port b

Letter argument transmits to only one port.
Otherwise, number specifies both port and channel

See Also

pgmin	Ouput incoming MIDI program change values
midiout	Transmit raw MIDI data
Ports	How ports are specified
Tutorial 16	More MIDI ins and outs

pics

Animation in graphics window

Input

int — Draws the specified frame number if the object's associated graphics window is visible. In a **pics** object, the first frame number is always 0. In addition, the "next frame" pointer is set to the frame number drawn.

bang — Draws the next frame. **pics** will cycle back to the first frame after drawing its last frame.

priority — The word priority, followed by a number greater than 0, sets the object's *sprite priority* to that number. Refer to the Graphics section of *Topics* for a discussion of sprite priorities.

Arguments

pics2 — Optional. If the object is created as **pics2**, the first frame is taken to be a background frame which is not erased by subsequent frames. Since the PICS file format is ambiguous on what the first frame does, you should try creating the object as **pics2** if you don't get the proper results with **pics**.

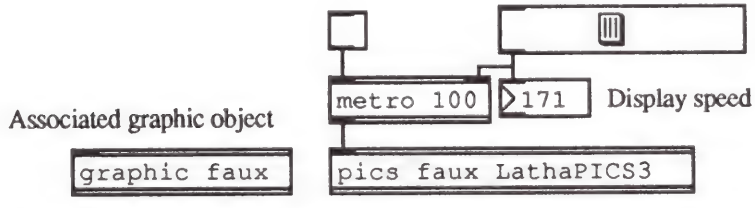
symbol — Obligatory. The first argument to **pics** must be a name of a **graphic** object whose window will be used to draw the animation. The second argument must be the name of a file in PICS format which will be loaded when the object is created. Note that the size of PICS files is often quite large, and loading the file may take a while.

int — Optional. Following the window name and file name, a number greater than 0 sets the inital sprite priority. The default priority is 3.

Output

(visual) — When the **pics** object's associated graphics window is visible, and a bang message or a number is received in its inlet, a frame is drawn in the window, and the object's previously drawn frame (if any) is erased, unless the object was created as **pics2**, in which case the first frame always stays on the screen.

Examples



Using a metronome to cycle through a PICS file at a constant speed.

Output incoming MIDI poly pressure values

polyin

Input

(MIDI) There are no inlets. The input is a MIDI polyphonic key pressure message received directly from the serial ports, or from the *virtual ports* of the MIDI Manager.

Arguments

a-f Optional. Specifies the port from which to receive incoming MIDI messages. If there is no argument, **polyin** receives from all channels on all ports.

a-f and int A letter and number combination (separated by a space) indicates a port and a specific MIDI channel on which to receive messages. Channel numbers greater than 16 will be *wrapped around* to stay within the 1-16 range.

int A number alone can be used in place of a letter and number combination. Numbers 1-16 specify a channel number on port a, numbers 17-32 specify channels 1-16 on port b, etc.

If the MIDI Manager is being used, a letter a-f specifies a virtual port. If the MIDI Manager is not being used, the letter a specifies the modem port and b specifies the printer port.

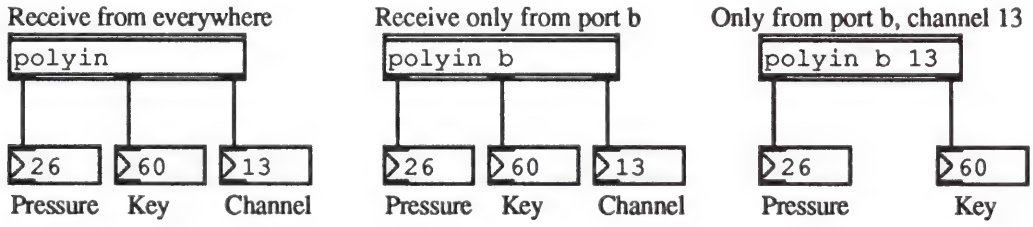
Output

int Out left outlet: The number is the pressure value of the incoming polyphonic key pressure message.

Out 2nd outlet: The number is the pitch value (key number) of the incoming message.

If a specific channel number is included in the argument, there are only two outlets. If there is *no* channel number specified by the argument, **polyin** will have a third outlet, on the right, which will output the channel number of the incoming note-on message.

Examples



Messages can be received from everywhere, a specific port, or a specific port and channel

polyin

Output incoming MIDI poly pressure values

See Also

polyout	Transmit MIDI poly pressure messages
midiin	Output incoming raw MIDI data
Ports	How ports are specified
Tutorial 16	More MIDI ins and outs

Transmit MIDI poly pressure messages

polyout

Input

int — In left inlet: The number is the pressure value of a MIDI polyphonic key pressure message transmitted on the specified channel and port. Numbers are limited between 0 and 127.

In middle inlet: The number is stored as the key number, to be used with pressure values received in the left inlet. Numbers are limited between 0 and 127.

In right inlet: The number is stored as the channel number on which to transmit the polyphonic key pressure messages.

float — Converted to int.

list — In left inlet: The first number is the pressure value, the second number is the key number, and the third number is the channel, of a transmitted MIDI polyphonic key pressure message.

port — In left inlet: The word port, followed by a letter a-f, specifies the port used to transmit the MIDI messages.

Arguments

a-f — Optional. Specifies the port for transmitting MIDI messages. Channel numbers greater than 16 received in the right inlet will be *wrapped around* to stay within the 1-16 range. If there is no argument, **polyout** initially transmits out port a, on MIDI channel 1.

a-f and int — A letter and number combination (separated by a space) indicates a port and a specific MIDI channel on which to transmit messages. Channel numbers greater than 16 will be *wrapped around* to stay within the 1-16 range.

int — A number alone can be used in place of a letter and number combination. Numbers 1-16 specify a channel number on port a, numbers 17-32 specify channels 1-16 on port b, etc. In this case, channel numbers received in the right inlet will specify both the port and the channel number.

If the MIDI Manager is being used, a letter a-f specifies a virtual port. If the MIDI Manager is not being used, the letter a specifies the modem port and b specifies the printer port.

Output

(MIDI) — There are no outlets. The output is a MIDI polyphonic key pressure message transmitted directly to the serial ports, or to the *virtual ports* of the MIDI Manager.

polyout

Transmit MIDI poly pressure messages

Examples

Letter argument transmits to only one port.
Otherwise, number specifies both port and channel

See Also

polyin	Ouput incoming MIDI polyphonic key pressure values
midiout	Transmit raw MIDI data
Ports	How ports are specified
Tutorial 16	More MIDI ins and outs

Selectively pass the input out a specific outlet

route

nput

anything — If the first item of the message is the same as one of the arguments of **route**, the rest of the message is sent out the outlet that corresponds to that argument. If the first item does not match any of the arguments, the entire message is passed out the rightmost outlet.

Arguments

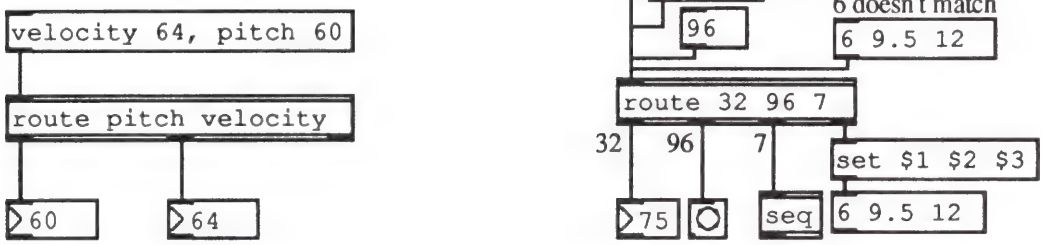
int, symbol — Optional. Arguments must be either all ints or all symbols. The number of arguments determines the number of outlets, *in addition to* the rightmost outlet. Each argument assigns a name or a number to an outlet. The maximum number of arguments is 10. If there is no argument, there is one other outlet, which is assigned the number 0.

Output

anything — The first item of any message received in the inlet is compared with the arguments. If it matches one of the arguments, the rest of the message is sent out the specified outlet. Otherwise, the entire message is passed out the rightmost outlet.

If the first item of a message matches one of the arguments, but the message has no additional items, bang is sent out the specified outlet.

Examples



Arguments assign names or numbers to the outlets, and route the input to the correct outlet

See Also

gate	Pass the input out a specific outlet
select	Select certain inputs, pass the rest on
Tutorial 17	Gates and switches

rtin

Output incoming MIDI real time messages

Input

(MIDI) There are no inlets. The input is a MIDI real time message received directly from the serial ports, or from the *virtual ports* of the MIDI Manager.

Arguments

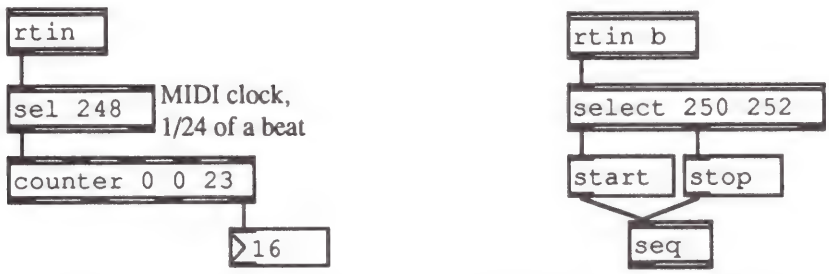
a-f Optional. Specifies the port from which to receive incoming MIDI real time messages. If there is no argument, **rtin** receives from port a.

If the MIDI Manager is being used, a letter a-f specifies a virtual port. If the MIDI Manager is not being used, the letter a specifies the modem port and b specifies the printer port. Note that if Filter Time Code is checked in the Apple MIDI Driver Settings window of the PatchBay application, MIDI clock messages will not be passed through to MIDI Manager applications. You may wish to use **timein** or an externally sync-ed **metro** or **clocker** in this case.

Output

int MIDI real time messages (MIDI clock, start, stop, and continue) received from the specified port are sent out the outlet.

Examples



MIDI real time messages can be used to synchronize Max with external events

See Also

clocker	Report elapsed time at regular intervals
MIDI	MIDI overview and specification
metro	Output bang at regular intervals
midiin	Receive raw MIDI data
timein	Report time from external time code source
seq	Sequencer for recording and playing MIDI
Tutorial 16	More MIDI ins and outs

Send messages,
without patch cords

send / s

Input

anything — A message received in the inlet is sent out the outlet of any **receive** object that has the same name, even if the **receive** is in another loaded patch.

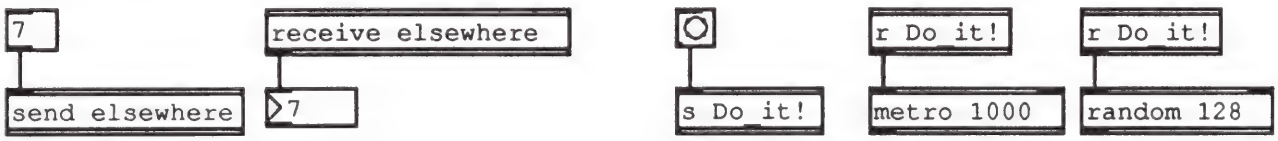
Arguments

any symbol — Obligatory. Gives a name to the **send** object.

Output

anything — There are no outlets. A message received in the inlet of **send** is sent out the outlet of any **receive** with the same name, even if the **receive** is in another loaded patch.

Examples



Virtual connections exist between all **send** *and* **receive** *objects which share the same name*

See Also

receive — Receive messages, without patch cords
value — Share a stored number with other objects
Tutorial 24 — **send** and **receive**

seq

Sequencer for recording and playing MIDI

Input

bang — Starts playing the sequence stored in **seq**.

start — The word start by itself has the same effect as bang. The word start, followed by a number, plays the stored sequence at a tempo determined by the number. The message start 1024 indicates normal tempo. If the number is 512, **seq** plays the sequence at half the original recorded speed, start 2048 plays it back at twice the original speed, and so on.

record — Starts recording MIDI messages received in the inlet.

stop — Stops the sequencer if it is recording or playing. A stop message need not be received when switching directly from playing to recording, or vice-versa.

append — Starts recording at the end of the stored sequence, without erasing the existing sequence.

int — When **seq** is recording, numbers received in its inlet are interpreted as bytes of MIDI messages (usually from **midiformat** or **midiin**). MIDI channel messages and system exclusive messages can be recorded by **seq**, but **seq** does not respond directly to MIDI real time messages such as start, stop, MIDI clock, etc.

float — Converted to int.

delay — The word delay, followed by a number, sets the onset time, in milliseconds, of the first event in the recorded sequence.

hook — The word hook, followed by a float, multiplies all the event times in the stored sequence by that number. For example, if the number is 2.0, all event times will be doubled, and the sequence will play back twice as slowly. Multiplications can even be performed while the sequence is playing.

write — Calls up the standard **Save As...** dialog box, so that a recorded sequence can be saved as a separate file. If you want to edit the sequence with the text editor, check the *Save As Text* option in the dialog box.

read — With no arguments, read calls up the standard **Open...** dialog box, so that a previously recorded sequence can be read into **seq**, replacing the current sequence. With a symbol as an argument, read searches for a file with the specified name to read into the **seq** object.

print — Prints the first few events of the recorded sequence in the Max window.

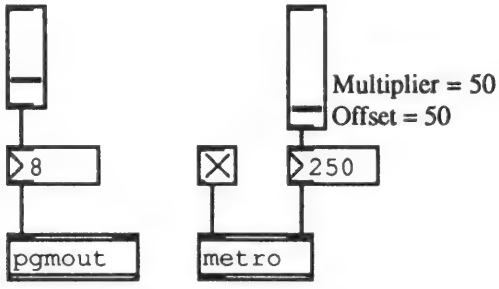
Arguments

any symbol — Optional. Specifies the name of a file to be read into **seq** automatically when the patch is loaded.

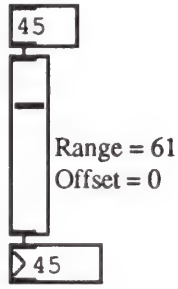
Output numbers by moving a slider

slider

Examples



Produce output by dragging onscreen...



or display numbers passing through

See Also

slider exists for compatibility with old Max files. **hslider** and **uslider** have many more features.

dial	Output numbers by moving a dial onscreen
hslider, uslider	Output numbers by moving a slider onscreen
Tutorial 9	Using the slider
Tutorial 14	Sliders and dials

speedlim

Limit the speed that numbers can pass through

Input

int — In left inlet: The number will be passed out the outlet, provided that a certain minimum time has elapsed since the previous output. Otherwise, the number is ignored.

In right inlet: The number is stored as the minimum amount of time, in milliseconds, between successive outputs.

float — Converted to int.

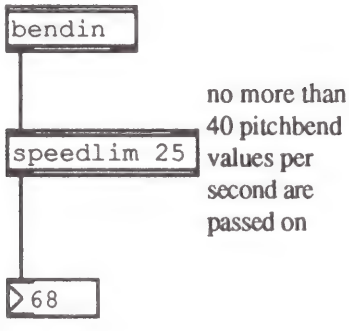
Arguments

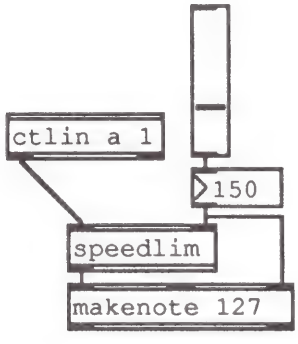
int — Optional. Sets an initial minimum time between outputs, in milliseconds. If there is no argument, the minimum time is 0.

Output

int — A number received in the left inlet is sent out the outlet, provided the specified minimum amount of time has elapsed since the previous output. Otherwise, the number is ignored.

Examples





Used to reduce a heavy flow of numbers, or to turn a continuous flow into discrete steps

See Also

mousefilter	Pass numbers only when the mouse button is up
timer	Report elapsed time between two events
Tutorial 16	More MIDI ins and outs

Output incoming MIDI system exclusive messages

sysexin

Input

(MIDI) There are no inlets. The input is a MIDI system exclusive message received directly from the serial ports, or from the *virtual ports* of the MIDI Manager.

Arguments

a-f Optional. Specifies the port from which to receive incoming MIDI system exclusive messages. If there is no argument, **sysexin** receives from port a.

If the MIDI Manager is being used, a letter a-f specifies a virtual port. If the MIDI Manager is not being used, the letter a specifies the modem port and b specifies the printer port.

Output

int MIDI system exclusive messages received from the specified port are sent out the outlet, byte-by-byte.

Examples

MIDI system exclusive messages can be interpreted by a **lib** *object*

See Also

MIDI	MIDI overview and specification
midiin	Receive raw MIDI data
lib	Script-driven patch librarian
Tutorial 34	Managing raw MIDI data

table

Store and graphically edit an array of numbers

Input

list In left inlet: The second number is stored in **table**, at the address (index) specified by the first number.

int In left inlet: The number specifies an address in the **table**. The value stored at that address is sent out the left outlet. *However*, if a value has been received in the right inlet, **table** *stores* that value in the specified address, rather than sending out a number.

In right inlet: The number specifies a value to be stored in **table**. The next address number received in the left inlet causes the value to be stored at that address.

float Converted to int.

send The word send, followed by the name of a **receive** object, followed by an address number, sends the value stored at that address to all **receive** objects with that name, without sending the value out the **table**'s outlet.

length In left inlet: Sends the length (size) of the **table** out the left outlet.

size In left inlet: The word size, followed by a number, sets the size of the **table** to that number.

set In left inlet: The word set, followed by a list of numbers, stores values in certain addresses. The first number after the word set specifies an address. The next number is the value to be stored in that address, and each number after that is stored in a successive address.

clear In left inlet: Sets all values in the **table** to 0.

goto In left inlet: The word goto, followed by a number, sets a pointer to the address specified by the number. The pointer is set at the beginning of the **table** initially.

next In left inlet: Sends the value stored in the address pointed at by the goto pointer out the left outlet, then sets the pointer to the next address. If the pointer is currently at the last address in the table, it *wraps around* to the first address.

prev In left inlet: Causes the same output as the word next, but the pointer is then decremented rather than incremented. If the pointer is currently at the first address in the table, it *wraps around* to the last address.

refer In left inlet: The word refer, followed by the name of another **table**, sets the receiving **table** object to read its values from the named **table**.

inv In left inlet: The word inv, followed by a number, finds the first value which is greater than or equal to that number, and sends the *address* of that value out the left outlet.

Store and graphically edit an array of numbers

table

sum — In left inlet: Sends the sum of all the values in the **table** out the left outlet.

quantile — In left inlet: The word quantile, followed by a number, multiplies the number by the sum of all the values in the **table**. This result is then divided by 2^{15} (32,768). Then, **table** sends out the address at which the sum of all values *up to* that address is greater than or equal to the result.

bang — In left inlet: Same as a quantile message with a random number between 0 and 32,768 as an argument.

(mouse) — The values stored in **table** can be entered and edited graphically with the mouse. When a **table** object is first created in a Patcher window, the **table**'s graphic editing window is opened, in which values can be entered by drawing with the mouse. The editing window provides a palette of graphic editing tools.

show crosshairs, to aid precision drawing

select a region to cut, copy, clear, or paste

drag the display, to see another part of the table

draw in values freehand with the mouse

draw values in a straight line, from click to click

zoom the horizontal display in or out

zoom the vertical display in or out

When the Patcher window is locked, the graphic editing window can be opened by double-clicking with the mouse on the **table** object .

A **table** can be created in a separate file by opening a new Table window and choosing the **Save** command from the File menu. A **table** can also be created in a separate file by opening a new Text file, and simply beginning the file with the word table. The word table should be followed by a list of space-separated numbers, specifying values to be stored in the **table**.

A **table** which has been saved as a file can be viewed and edited as text by choosing **Open as Text...** from the File menu. Numbers in the form of text can be pasted in from other sources such as the editing window of a **capture** object, or even from another program such as a word processor. Text from a **capture** object can also be pasted directly into **table**'s graphic editing window.

(Get Info...) — When the editing window is open, various traits of the **table** can be modified by choosing **Get Info...** from the Max menu.

Size determines the number of values stored in the **table**. A newly created **table** has 128 values, indexed with numbers from 0 to 127.

table

Store and graphically edit an array of numbers

Range determines the range of values which can be displayed on the *y* axis of the editing window. A newly created **table** has a range of 128, from 0 to 127.

If *Signed* is checked, **table** displays negative numbers as well as positive. In effect, the range of displayed values specified by *Range* is doubled when the *Signed* option is checked, since the range goes in both directions from 0.

If *Note Name Legend* is checked, values are shown on the *y* axis as MIDI note names, rather than numbers.

If *Don't Save* is checked, Max will not ask if you want to save changes made to the **table**, when the patch containing that **table** is closed.

If *Save with Patcher* is checked, the values in the **table** are saved as part of the patch that contains it. Otherwise, the **table** has to be saved in a separate file to retain its values.

Arguments

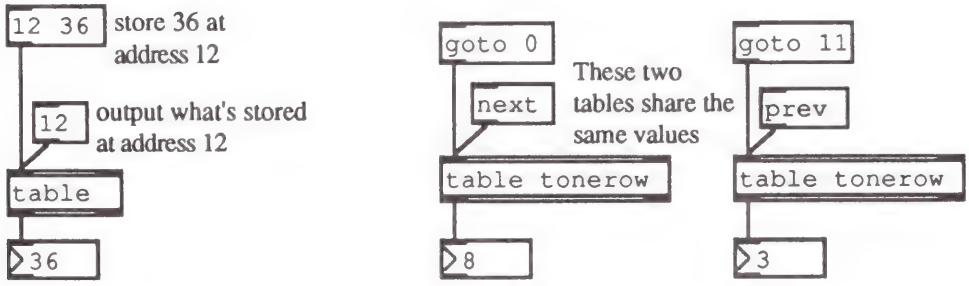
any symbol — Optional. The argument gives a name to the **table**. Max looks for a **table** of the same name which has been saved as a separate file. If two or more **table** objects share the same names, they also share the same data, even if Max couldn't find a file with the name.

Output

int — All numbers sent out by **table** are sent out the left outlet.

bang — When the contents of a table have been changed by an edit in the graphic editing window, bang is sent out the right outlet.

Examples



An array of any size and range can be stored, recalled, and modified

Store and graphically edit an array of numbers

table

See Also

capture	Store numbers to view or edit
coll	Store a collection of different messages
Data Structures	Ways of storing data in Max
funbuff	Store x,y pairs of numbers together
Histo	Make a histogram of the numbers received
Quantile	Using **table** for probability distribution
Tables	Using the **table** graphic editing window
Text	Format numbers as a text file
Tutorial 32	The **table** object

tempo

Output numbers at a metronomic tempo

Input

bang — In left inlet: Starts **tempo**'s metronome process, or *re*starts it if **tempo** is already on.

stop — In left inlet: Stops **tempo**.

int — In left inlet: If the number is not 0, it has the same effect as bang. If the number is 0, it has the same effect as stop.

In 2nd inlet: The number is stored as the tempo, in beats per minute (quarter notes per minute). The tempo is limited between 5 and 300 beats per minute.

In 3rd inlet: The number is a *beat multiplier*, which can lengthen the amount of time taken for one beat. It slows the tempo down by a factor. For example, a multiplier of 2 will make **tempo** send out its output half as fast.

In right inlet: The number is the rhythmic value sent out by **tempo**, specified as a fraction of a whole note. For example, the number 8 causes **tempo** to output eighth notes, relative to the specified (quarter note) tempo. The numbers sent out the outlet cycle continuously between 0 and the number 1 less than the rhythmic value. The divisions of a whole note must be between 1 and 96.

float — Converted to int.

tempo — In left inlet: The word tempo, followed by a float, sets the current tempo to the number, allowing tempos to be expressed in floating-point.

Arguments

int — Optional. The first argument sets an initial tempo, from 5 to 300 beats per minute. If there is no argument, the initial tempo is 120 beats per minute. The second argument is the beat multiplier and is always set to 1. The third argument sets an initial rhythmic value of the output, from a whole note (1) to a 64th note triplet (96). If the argument is not present, the initial value is 16.

Output

int — When **tempo** is started it outputs numbers in a continuous cycle from 0 to the number 1 less than the specified rhythmic value. The speed at which the numbers are sent out is determined by the tempo (quarter note beats per minute) and the rhythmic value of the output (fraction of a whole note).

Output incoming MIDI aftertouch values

touchin

Input

(MIDI) There are no inlets. The input is a MIDI aftertouch (channel pressure) message received directly from the serial ports, or from the *virtual ports* of the MIDI Manager.

Arguments

a-f Optional. Specifies the port from which to receive incoming aftertouch messages. If there is no argument, **touchin** receives from all channels on all ports.

a-f and int A letter and number combination (separated by a space) indicates a port and a specific MIDI channel on which to receive aftertouch messages. Channel numbers greater than 16 will be *wrapped around* to stay within the 1-16 range.

int A number alone can be used in place of a letter and number combination. Numbers 1-16 specify a channel number on port a, numbers 17-32 specify channels 1-16 on port b, etc.

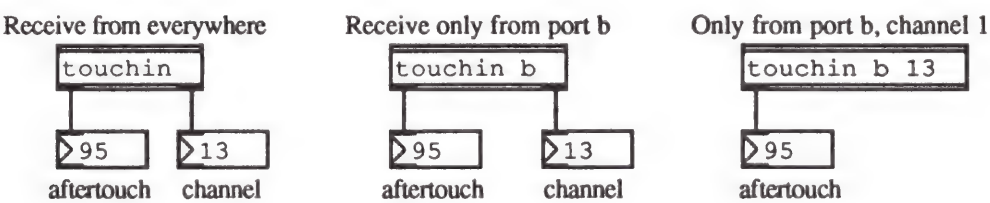
If the MIDI Manager is being used, a letter a-f specifies a virtual port. If the MIDI Manager is not being used, the letter a specifies the modem port and b specifies the printer port.

Output

int If a specific channel number is included in the argument, there is only one outlet. The output is the incoming aftertouch value, from 0-127, on the specified channel and port.

If there is no channel number specified by the argument, **touchin** will have a second outlet, on the right, which will output the channel number of the incoming aftertouch message.

Examples



Aftertouch messages can be received from everywhere, a specific port, or a specific port and channel

touchin

Output incoming MIDI aftertouch values

See Also

touchout	Transmit MIDI aftertouch messages
midiin	Output incoming raw MIDI data
Ports	How MIDI ports are specified
Tutorial 16	More MIDI ins and outs

Transmit MIDI aftertouch messages

touchout

Input

int — In left inlet: The number is transmitted as an aftertouch value on the specified channel and port. Numbers are limited between 0 and 127.

In right inlet: The number is stored as the channel number on which to transmit the aftertouch messages.

float — Converted to int.

list — In left inlet: The first number is the aftertouch value, and the second number is the channel, of a MIDI aftertouch message, transmitted on the specified channel and port.

port — In left inlet: The word port, followed by a letter a-f, specifies the port used to transmit the MIDI messages.

Arguments

a-f — Optional. Specifies the port for transmitting MIDI aftertouch messages. Channel numbers greater than 16 received in the right inlet will be *wrapped around* to stay within the 1-16 range. If there is no argument, **touchout** initially transmits out port a, on MIDI channel 1.

a-f and int — A letter and number combination (separated by a space) indicates a port and a specific MIDI channel on which to transmit aftertouch messages. Channel numbers greater than 16 will be *wrapped around* to stay within the 1-16 range.

int — A number alone can be used in place of a letter and number combination. Numbers 1-16 specify a channel number on port a, numbers 17-32 specify channels 1-16 on port b, etc. In this case, channel numbers received in the right inlet will specify both the port and the channel number.

If the MIDI Manager is being used, a letter a-f specifies a virtual port. If the MIDI Manager is not being used, the letter a specifies the modem port and b specifies the printer port.

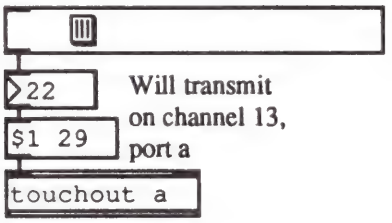
Output

(MIDI) — There are no outlets. The output is a MIDI aftertouch message transmitted directly to the serial ports, or to the *virtual ports* of the MIDI Manager.

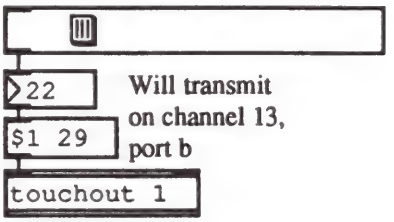
touchout

Transmit MIDI aftertouch messages

Examples



Letter argument transmits to only one port



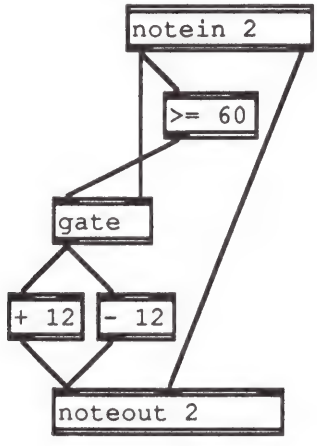
Otherwise, number specifies both port and channel

See Also

touchin	Ouput incoming MIDI aftertouch values
midiout	Transmit raw MIDI data
Ports	How ports are specified
Tutorial 16	More MIDI ins and outs

greater than or equal to 60, the **>=** object sends out 1, causing the **sel** object to trigger the messages -12 and 12. These two numbers are added to the original pitch, and the sums are sent to **noteout**, where they are combined with the velocity of the played note, triggering note messages to the synth. This process occurs both for note-ons and note-offs, so the transposed pitches are successfully played and turned off.

In Solution 2, a successful comparison opens a **gate**, letting the pitch through, so that only pitches greater than or equal to 60 will be passed on.



The placement of the **>=** object is of utmost importance. Because it is to the *right* of the **gate** object, it receives the pitch first, and either opens or shuts the **gate** before the pitch arrives at the **gate**.

When a pitch is let through, it has 12 subtracted from it in one object, and has 12 added to it in another object. The results are sent to **noteout** as pitches, where they are combined with the original played velocity (both for note-ons and note-offs).

Note: The **notein** and **noteout** objects in Solution 2 are set to receive and transmit on MIDI channel 2 only so that a note from your MIDI keyboard won't cause *both* patches to react.

Summary

In this exercise, the comparison **>=** 60 was used to trigger messages in one case, and to open a **gate** in another case. Either method could be incorporated in a solution to the problem.

To perform a test and make a decision, ask yourself these questions:

1. What do I need to know to make the decision? (What will be tested?)
2. What action will be taken if the test is successful (true)?
3. What action will be taken if the test is unsuccessful (false)?

Types and Declarations

expr uses some special data structures to communicate with your function. The arguments and result are of type Ex_ex, similar (but not identical) to an Atom. Ex_ex is defined as follows:

```
typedef struct ex_ex {
    union {
        long v_int;
        float v_flt;
        long op;
        char **ptr;
    } ex_cont;          /* content */
    long ex_type;       /* type of the node */
} Ex_ex;
```

The following macros are used to shorten the access for particular elements:

```
#define     ex_int      ex_cont.v_int
#define     ex_flt      ex_cont.v_flt
#define     ex_op       ex_cont.op
#define     ex_ptr      ex_cont.ptr
```

You'll need to check the `ex_type` field of an Ex_ex to determine the type of the element, then use the proper access macro. The possible types are defined as follows:

```
#define     ET_INT      0x1         /* an int */
#define     ET_FLT      0x2         /* a float */
#define     ET_OP       0x3         /* operator */
#define     ET_STR      0x4         /* string */
#define     ET_TBL      0x5         /* a table, the content is a pointer */
#define     ET_FUNC     0x6         /* a function */
#define     ET_SYM      0x7         /* symbol ("string") */
#define     ET_VSYM     0x8         /* variable symbol ("$s?") */
```

For almost every situation, you'll be dealing with `ET_INT` and `ET_FLT` types. You should probably do nothing (or return an error) rather than deal with a type other than what you're expecting. It's unlikely that you'll be passed anything but integers and floats (in fact, the implementation of **expr** would have to change radically before this will happen).

Declare your function as:

```
long main (argc, argv, optr)
long argc;
Ex_ex *argv,*optr;
{

}
```

`argc` is the number of arguments you'll receive (which you specified by numbering your **Expr** resource). Notice that unlike other Max calls, it is passed as a long. `argv` is a pointer to an array of the arguments passed to your function (from left to right), and `optr` is a pointer to an Ex_ex where you should place the result of evaluating your function.

Screen aesthetics

Tutorial 19

When editing a patch, you can select objects and/or patch cords and choose **Hide On Lock** from the Max menu. This sets those objects to be invisible when the Patcher window is locked. **Show On Lock** makes objects visible which were previously made invisible with **Hide On Lock**.

Hide On Lock is an invaluable feature for making patches with a good user interface. For example, in the patch in the right part of the Patcher window, there is no reason that the user needs to see the **pgmout** object and the patch cords connecting the various objects. All that the user of your patch needs or wants to see is the **dial**, the label (**comment**), and the number being sent out by the **dial**.

The Hide On Lock command lets you hide unsightly parts of a patch

Note: If you select a region of the Patcher window and hide several objects at once with **Hide On Lock**, the objects will be hidden but the patch cords will still be visible. To hide patch cords, you must select them before choosing **Hide On Lock**. You can select patch cords by holding down the option key while dragging a box around a group of objects. Alternatively, choosing **Select All** from the Edit menu selects all boxes *and* patch cords, or you can just click on patch cords individually, using the shift key to select multiple objects.

Graphical user interface objects (such as **button**, **toggle**, **dial**, etc.) which have been hidden with **Hide On Lock** do not respond to clicking and dragging with the mouse.

Paste Picture

Your completed patch can actually have *any* visual appearance you choose, because you can import pictures from other applications such as painting and drawing programs. Cut or Copy the picture from another application, then in Max, choose **Paste Picture** from the Edit menu to paste the picture into a Patcher window.

For example, the decorative border around the **dial** in this patch is actually just a frame drawn in a painting program. The **dial** and the frame were then placed on top of each other to give the illusion of a different kind of **dial**.

The following example (which is already provided as a built-in **expr** function min) computes the minimum of two numbers. Just to keep things simple, it doesn't perform any initialization, use any global variables, or access Max function macros. It *does* show how to access and type-check function arguments and return a result.

```
long main(argc, argv, optr)
long argc;
register Ex_ex *argv, *optr;
{
    register Ex_ex *left, *right;

    if (argc == -1)    /* REQUIRED check for initialization time */
        return;

    left = argv++;
    right = argv;
    if (left->ex_type == ET_FLT) {
        if (right->ex_type == ET_FLT) {
            if (left->ex_flt < right->ex_flt)
                *optr = *left;     /* copy arg -> result, incl. type */
            else
                *optr = *right;
        } else {
            if (left->ex_flt < (float)right->ex_int)
                *optr = *left;
            else
                *optr = *right;
        }
    } else {
        if (right->ex_type == ET_FLT) {
            if ((float)left->ex_int < right->ex_flt)
                *optr = *left;
            else
                *optr = *right;
        } else {
            if (left->ex_int < right->ex_int)
                *optr = *left;
            else
                *optr = *right;
        }
    }
}
```

The following example calls the built-in C function pow(), converting its arguments to double before calling the function, and converting the result of pow back to an integer if both arguments are integers, or float otherwise. Note that you would have to include the Math library in your code resource to call pow(). Since all of the Math library's functions are already available in **expr** anyway, this is a pretty hypothetical example.

Presets

Max contains a special **preset** object, which has the ability to send preset messages to some or all of the objects (*clients*) in a Patcher window. The preset message, sent when the user is *storing* a preset, is just a request for your object to tell the preset object how to restore your internal state to what it is now. Later, when the user *executes* a preset, the **preset** object will send you back the message you had previously specified.

The dialog goes something like this:

• During a store...

preset *object object to Client object(s)*: hello, this is the preset message—tell me how to restore your state

Client object to **preset** *object:* send me int 34 (for example)

• During an execute...

preset *object to Client object*: int 34

The client object won't know the difference between receiving int 34 from a **preset** object and receiving a 34 in its leftmost inlet.

It's not mandatory for your object to respond to the preset message, but it is something that will make users happy. All Max user interface objects currently respond to preset messages.

Here are routines you can use when responding to the preset message:

```
preset_store (char *format, void items…)
```

This is a general preset function for use when your object's state cannot be restored with a simple int or set message. The arguments are essentially the same as those you would pass to `binbuf_vinsert` (see above). The example below shows the expected format for specifying what your current state is to a **preset** object. The first thing you supply is your object itself, followed by the symbol which is the name of your object's class (which you can retrieve from your object using the macro `ob_sym`, declared in `ext_mess.h`). Next, supply the symbol which specifies the message you want receive (a method for which had better be defined in your class), followed by the arguments to this message—the current values of your object's fields.

Here's an example of using `preset_store` which specifies that the object would like to receive a set message. We assume it has one field, `myvalue`, that it would like to save and restore.

```
void *myobject_preset(x)
struct myobject *x;
{
    SetUpA4();

    preset_store("ossl",x,ob_sym(x),gensym("set"),x->myvalue);

    RestoreA4();
}
```

Tutorial 9

Using the **slider**

Summary

A **slider** lets you output a continuous stream of numbers within a specified range by dragging on it with the mouse. It will also show and send out numbers received in its inlet, making it useful for graphically displaying the numbers passing through it.

By choosing **Get Info...** from the Max menu, you can change the *Slider Range*, and can also specify a *Multiplier*, by which all numbers will be multiplied before being sent out, and an *Offset*, which will be added to the number, after multiplication.

See Also

slider	Output numbers by moving a slider

2.5 Addendum

About this Document

The Max 2.5 manual addendum describes the new features in Max versions 2.2 and 2.5. It's been formatted so you can insert the new pages updating Topics and Objects sections into your existing (and bulging) Max manual. In addition, there is a section at the beginning which covers general new features and minor changes to Max objects.

In addition to the printed documentation, there are additional files in Microsoft Word format included in version 2.5. One describes an experimental object called **pr** that adds deeper object-oriented programming capabilities to Max. Several others update the C developer documentation. In addition, we've included the latest version of "Writing External Objects for Max" since many developers have requested it.

Installation

The Max 2.5 Update Package contains six disks.

Disk 1 contains the copy-protected Max application, Max Object List file (see below), and the Max Tutorial folder.

Disk 2 contains the max-help and max-startup folders.

Disk 3 contains the patches folder, C developer materials, and Unsupported Objects

Disk 4 contains Applications.

Disk 5 contains more Applications.

An additional disk contains OMS (the Opcode MIDI System) and the MIDI Manager.

It is really *essential* that you replace all your old help files, startup files, and other external objects with the new files which come with version 2.5. Remember, Max consists of more than an application program: the external objects and help files make up an important part of the Max "content."

- If you have the space on your hard disk, leave your existing Max folder on the disk.
- Create a new folder called Max (or Max 2.5 if you've kept your old Max folder).
- From Disk 2, copy the max-startup and max-help folders to your Max folder.
- From Disk 3, copy the patches folder to your Max folder. If the other folder on Disk 3 are of interest to you, copy them to your hard disk.
- Insert Disk 1, and double-click on the Max 2.5 application on the floppy disk, and follow the instructions in the dialog boxes to hard disk install a copy of Max into your new Max folder.
- If applicable, copy any files you want from your old Max folder into the Max 2.5 folder.
- The Applications disks can be installed by inserting each disk and dragging its icon to your hard disk. The Finder may inform you about creating a folder on your hard disk.

Applications

Max 2.5 includes a few new applications.

Procussion Editor/Librarian

The long-awaited editor/librarian for the Emu Procussion is included with this release of Max. Those interested in writing librarians and editors may wish to tear the application apart to see how it makes use of some of the newer features of Max.

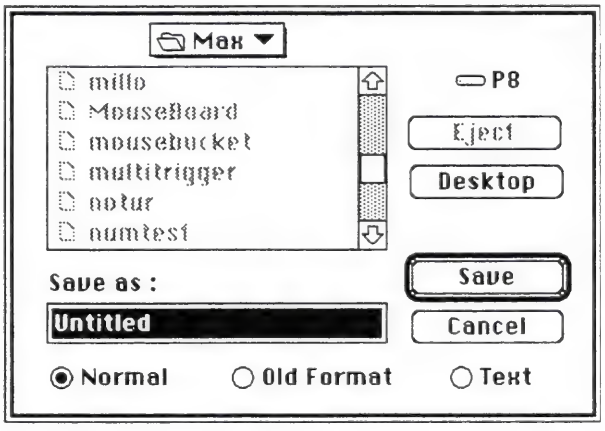
MacSoutiLs from INA + GRM

Opcode is pleased to distribute a set of patches from the *Groupe de Recherches Musicales* (GRM) in Paris. These elegant applications share a common interface. They are intended for both composition and a somewhat new category of "MIDI Sound Effect" generation.

2.5 Features

New File Format

Max has a new file format, which produces files which are usually 40-50% smaller than the previous format. File reading (but not saving) has been optimized in version 2.5—some informal tests indicate that large files will over four times faster in the new format. In addition, the new format offers optional (limited) protection against people unlocking your patcher. The new standard save file dialog is shown below.



Normal is the file format used by versions 2.2 and later of Max. The *Old Format* saves files readable by version 2.1 and earlier. Or you can save your file as *Text*, which will always be readable by any version of Max.

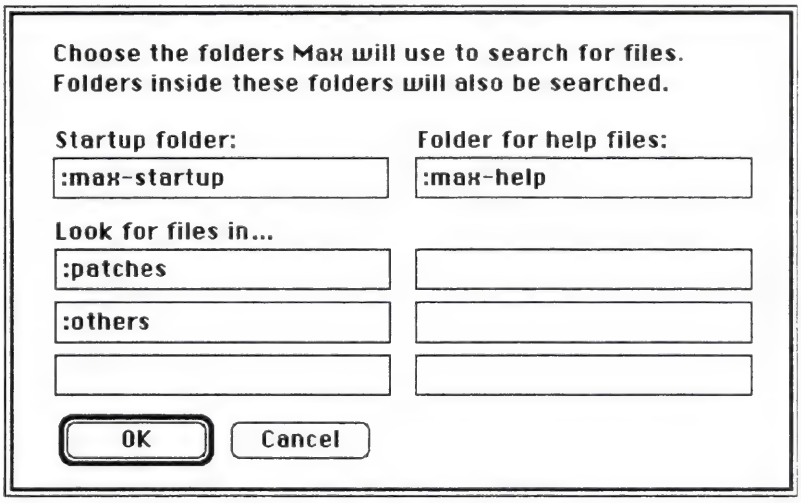
To protect a file when saving in the new format (which you choose as "Normal" in the Save As... dialog), hold down the option key when the file is actually being saved. You will hear two beeps if the file is being saved in the protected format. But wait...after a file is protected, it can never be unlocked—by anyone. So you never want to make your master copy of a patcher you're working on into a protected file.

If you read in a file which was in the Old Format or Text, it will retain that characteristic when you save it, although you can choose Save As... from the File menu to change its format.

2.5 Features

File Preferences and the Help Folder

The File Preferences dialog (documented in the 2.1 manual on pages 231-232) has been improved. The new dialog is shown below.



There are now six folder names which can be specified for searching for Max files. But more importantly, and folder inside these six folders will be searched as well, which means you can have an unlimited number of folders.

Folders inside the Help Folder (max-help) are searched as well, and we have taken advantage of this feature to reorganize the Help Folder. There are now sub-folders for specific types of Max objects and their related files. These folders have names like Graphics Help, Librarian Help, and DSP Help.

Folders inside the Startup Folder (max-startup) are not searched, because many users created folders inside the Startup Folder for external objects and patchers they didn't want loaded at startup.

File Aliases with System 7

You can make aliases to subpatchers and external objects, as well as folders that are searched by Max (as specified in the File Preferences dialog).

Aliasing a folder makes it easier to specify searching for objects on another volume. An easy way to use this feature is to make an alias of the folder you want searched, then drag it into one of the named folders you've specified to search in the File Preferences dialog.

Aliasing an external object can be used for externals that can be created under multiple names. For example, you might create three aliases for the **oval** object called rect, ring, and frame. Then the **oval** object can be taken out of the Startup Folder because typing any of its "aliases" will direct Max toward the oval external object. This "alias" problem only affects getting the **oval** object installed into Max; once its in memory, Max knows about the other names.

2.5 Features

Unfortunately, we can't make aliases for the **oval** object, because aliases remember the volume they were on, and you won't have access to a hard disk in a cubicle somewhere at Opcode. So you'll have to make the aliases for objects with multiple names yourself. We recommend placing the following alias files in the patches folder:

aliases for **oval**: rect, ring, frame
alias for **xbendin**: xbendin2
alias for **xbendout**: xbendout2
alias for **pics**: pics2

Order of Searching for Files

When the name of a patcher or data file is given to Max to open, the first folder searched is the one which contains the "top level" file being opened, or if no file is being opened, the first folder searched is the one containing the Max application.

Next, the folders listed in the File Preferences dialog are searched. Folders listed in the first column of the dialog box are searched first, then those listed in the second column (if any). The folders inside of each folder listed in the File Preferences dialog are searched as well, and each listed folder's sub-folders are all searched before the next one in the list is searched. In other words, Max conducts a depth-first search of folders.

Finally, if the first folder searched was not the folder containing the Max application, and the file still hasn't been found, the folder containing the Max application is searched.

New Option: Max Window Visible At Startup

A new item in the Options menu lets you decide whether you want to see the Max window when Max starts up. On the other hand, you now have no choice as to whether you see the Max About Box when the program starts up. Well, that's not strictly true. If you're a motivated ResEditor, look for a resource whose type might explain why the Max About Box appears each time the program starts up, and eliminate it. If you pick the right one, you might be pleasantly surprised. If not, you might ruin your copy of Max.

Max on the Clipboard

If you copy all or part of a patcher to the Clipboard, Max saves it in both PICT and binary format. This allows you to use the Scrapbook or similar program to create a catalog of Max snippets. A Scrapbook File with some examples has been included in the Scrapbook Snippets folder.

A "Max Object List" Made For You, Plus New Categories

Covered on pages 239-240 of the Max 2.0 manual, and 241-242 of the Max 2.1 manual.

Included on Disk 1, the text file "Max Object List" can be placed in the same folder as the Max application. This "standard" version of the file contains the names of all of the external objects you might want to type into an object box. When Max starts up, these names and associated categories are added to the New Object List window.

2.5 Features

While the ability to read the Max Object List file has existed for some time, no one had taken the time to include a file listing the external objects that ship with Max. You can modify this file to add your own external objects or Max patchers, or remove external objects you'll never use. The format of the file is explained on page 242 of the Max 2.1 manual (page 240 of the Max 2.0 manual).

The list of file categories has been changed. Two new categories have been added, and one has been renamed. The new categories are Devices, for control of non-MIDI hardware, and Messages, for objects that build or change Max messages. Keyboard/Mouse has been renamed Interaction, to cover a broader range of input devices.

Patcher Window Zooms, Uses System 7 Colors

The patcher window now has a zoom box, and when using System 7 in color, the window title bar and grow icon are in the correct colors. With narrow patcher windows, the lock and enable gadgets may disappear so that more of the title can be seen.

Color Objects and the Color Dialog

A new item in the Max menu, Color... is enabled when an object that can change its color is selected. When you choose Color... from the Max menu a dialog box with sixteen beautiful shades appears.

The dialog box will try to position itself below the item(s) you have selected, and allows you to click on different colors and see what the object will look like without closing the window. Click OK to change the color, or Cancel to revert to the old color.

The kernel user interface objects and standard external objects which support changes in color are the object box, **button**, **dial**, **hslider**, and **uslider**. The **led** object can appear in different colors but these are based on the choice of different pictures in its Get Info... dialog.

Note that patchers that contain objects which are changed to a color different from the default medium gray may produce error messages when read into previous versions of Max, but should still work properly.

Get Info... with No Objects Selected

When a patcher window is unlocked, and no boxes or lines are selected, choosing Get Info... from the Max menu prints (in the Max window) a list of external objects and subpatchers used in the patcher window and any of its subpatchers. This is useful for making a note of what files you need to copy in order to give someone else your Max patcher.

Minor Changes To Objects

buddy

The limitation of 10 inlets and outlets in a **buddy** object has been removed. In addition, **buddy** takes the clear message to reset its inlets.

2.5 Features

led

The state of **led** can now be stored in a preset. In addition, **led** accepts the pict message (which takes an integer argument) changing the picture used to in the object. With an unmodified **led**, this has the effect of changing the color used by the object.

The **led** object can be modified to use other pictures than the standard pictures of a diode. See the "modifying the **led** object" folder in the C development materials folders.

patcher

You can now type in more than one argument to a **patcher** object and all the text will appear in the title of a subpatcher window. In other words, if you make an object called **patcher function control**, its window title will no longer be only [function].

preset

The **preset** object's box can now be resized in both the X and Ydimensions, and the maximum allowable presets in one object has been increased to 2048. The **preset** object has an additional outlet which reports the preset number being executed.

Choosing Get Info... from the Max menu on a selected **preset** object opens a dialog box that allows you to type in an "ID Name"—the first four characters of this name, if you enter one (it's optional) are used as the Macintosh "file type" for files of presets saved by this object. When you send the read message to a **preset** object which has an ID Name, only the files whose types match the first four characters of this name are shown in the standard file dialog. This allows you to create a "document type" for preset files so the user won't open a preset file designed for another **preset** object (producing a string of error messages and hard feelings).

On page 416 of the manual (page 392 in the 2.0 manual), we claimed that you can connect the left outlet of a preset object to the inlet of a **send** object, but until version 2.5, this wasn't actually possible. The change to allow this was actually made in the **send** object.

print

Double-clicking on any **print** object opens the Max window or brings it to the front.

send
receive
value

It has never been previously documented, but double-clicking on a **send**, **receive**, or **value** object opens (in a somewhat cyclic fashion) windows which contain **send**, **receive** or **value** objects connected to the one you double-clicked.

seq

The **seq** object contains an additional outlet which sends out a bang message when the sequence is done playing.

The **seq** object reads and writes a text file format as well as standard MIDI files. The format of the text file is as follows. Each line consists of three or four numbers. The first number is the start time from the beginning of the file, in milliseconds. The next two or three numbers are the MIDI data located at that time. For example, here's the text file for a sequence which plays a middle C for 1 second.

0 144 60 64
1000 128 60 0

144 is the MIDI code for "note-on, channel 0" and 128 is the code for note-off. See the MIDI topic in the manual for more information.

serial

The **serial** object is now compatible with the GreenSpring Model 1280 Nubus serial card. Ports on a 1280 card can be specified as c, d, e, or f in the first argument to the **serial** object.

ubutton

The **ubutton** object has some new features useful for its toggle mode. First, the state of a **ubutton**, when in toggle or stay mode, is saved in a preset. **ubutton** also responds to the set message. set 1 sets the toggle and highlights the button, set 0 clears the toggle and unhighlights the button. You can use the set message to change the **ubutton**'s value to other non-zero numbers; this is useful in conjunction with stay mode.

New Unsupported Objects

aebang	Receive an Apple Event
andGate	Logical and bitwise and of multiple inputs (by David Roach)
orGate	Logical and bitwise or of multiple inputs (by David Roach)
lcd	Draw text and lines in a patcher window (by Michael Lee)

New Script File Messages for the lib Object:

aspatches — The aspatches script message takes an optional argument which specifies the first patch number to use when loading or sending a bank—this is for devices that number their modifiable memory at some non-zero value like 64.

Bulk Dumps of Packets

aspackets — The aspackets script message specifies that a device sends and receives its bulk dumps in packets of a fixed size, with the last packet possibly being of a smaller size. An argument specifies the packet size. The number of packets is determined by dividing the packet size by the size specified in the argument to the data keyword inside an inbank or outbank statement. To use the aspackets mode, you should write your inbank and outbank statements as if they were for sending out a single packet, but set the number of bytes of data to the total count of data in all the packets.

Two keywords have been added for use in MIDI communication statements in conjunction with the aspackets mode. These keywords are only useful for outbank statements; they may not work properly (and wouldn't do anything anyway) if placed in reception statements. If an incoming packet contains a packet oriented element, just use the dc (don't care) keyword for each byte.

address — The address keyword takes two arguments. The first is a starting address value, and the second is the number of bytes used by the address in the system exclusive message. The address is then incremented by the number of data bytes sent in each system exclusive packet, and inserted in the spot occupied by the address keyword in the MIDI message. For example, to use the "Roland Hex" address 48 00 00, you would use:

```
address 0x120000 3
```

size — The size keyword takes one argument, a number of bytes occupied by the size in the system exclusive message. The size of the packet being sent is inserted in the spot occupied by the size keyword in the MIDI message.

Checksums

There is now an additional checksum method, number 3, which includes the sum of any address bytes included in a MIDI message as well as the data bytes in a twos-complement sum.

ckstart — A new keyword in output communication statements, ckstart, does not send any data but instead informs the **lib** object to begin a checksum of any bytes it sends out. This sum will be terminated by a normal checksum statement. If no ckstart statement is present, the checksum will be made only of the "data" portion of the MIDI output. Here's an example of using ckstart to include the "address" into the checksum.

```
#L outbank 0xF0 0x44 0 / ckstart / address 0x120000 3 / data 128 / checksum 2 / 0xF0;
```

Variable-Length Data

variable — The variable script message allows the data keyword in an inbank or inpatch statement to specify the *maximum* number of bytes expected. If an F7 is seen before the end of the data, the **lib** object will consider the data completely received.

timeout — aspatches mode can work in conjunction with the timeout script message, whose argument specifies a time to wait for the next patch to be received. If no patch appears within the time allotted, the bank is considered received.

Miscellaneous

wordaccess — The wordaccess script message makes the **libfrom**, **libto**, and **libtofrom** objects refer to 16-bit offsets inside a patch instead of byte offsets, as well as peek and poke full 16-bit values in the patch's data buffer.

delay — The delay message can either be sent to **lib** via its left inlet or included in a script. The argument sets the number of milliseconds that **lib** waits between sending patches in aspatches mode.

id — The id message takes two arguments. The first specifies the lowest number of the ID numerical in the librarian window. Some devices display their device ID starting at 0 on the front panel, but the **lib** object's window used to display the ID only as a number between 1 and 16. Internally, the lowest ID always starts at 0; this first argument is just for display purposes.

The second argument specifies the number of possible device IDs accepted by the device. For example, many Roland synthesizers accept device IDs between 1 and 32, so an appropriate id message for such a device would be id 1 32.

The id message is optional in a librarian script, if not present, the ID numerical will start at 1 and have a maximum value of 16.

openprompt — The openprompt message uses the text that follows it for the directions displayed in the standard open file dialog box when opening a bank of patches. For example, you might wish to use openprompt as follows:

```
#L openprompt Open a File of DX7 Patches;
```

saveprompt — The saveprompt message uses the text that follows it as the directions displayed in the standard save file dialog box when saving a bank of patches. For example, you might wish to use saveprompt as follows:

```
#L saveprompt Name Your DX7 Patches;
```

New Features in Arguments to Editor Objects

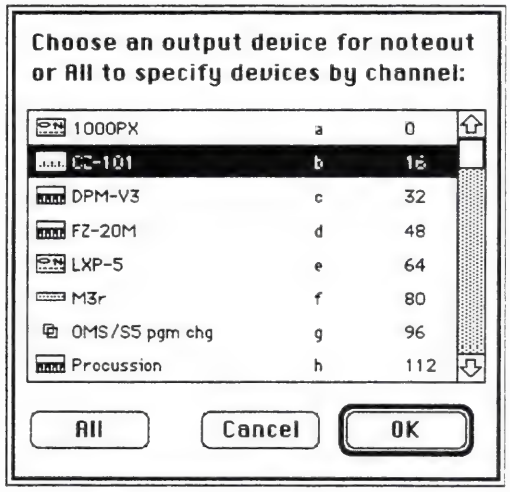
These changes add new features to the **libfrom** and **libto** objects, as well as the new **libtofrom** object.

and — The and argument keyword is used to get or set two bytes as a single number. For example **libfrom 4 and 5** outputs a single value whose most significant byte is located at index 4 and whose least significant byte is at 5. **libto 4 and 5** will set byte offset 4 with the most significant byte of its input, and byte offset 5 with the least significant byte. The combination of the two bytes is also the value of $i2 in a parameter change (param) message.

2s — The 2s argument keyword specifies that the parameter should be treated as a twos-complement number, which means that negative numbers can be sent to and received from the editor object. One number is required after 2s which specifies the number of bits used to represent the number. For example, 2s 7 specifies that seven bits are used (this is quite common), so values can range from -63 to +64. It is up to you to limit the number sent to **libto** and **libtofrom** objects. 2s can be combined with the and keyword. For example, in the Emu Procussion editor, parameters are stored as 14-bit numbers made out of two 7-bit numbers, and the entire 14-bit quantity is used to represent a negative number in twos-complement form. A typical specification for an editor object in the Procussion editor would be **libtofrom 4 and 5 p 30 2s 7**.

OMS Choose Device Dialog

When using OMS, double-clicking on any MIDI object opens a dialog box for choosing an OMS device. You can also bring up this dialog by sending any MIDI object the open message.



MIDI object Choose Device dialog box

The **midiin**, **midiout**, **sysexin**, and **rtin** objects force you to choose a single device, since they are designed to work with a single port at a time. The other input objects allow you to click All to connect the object to all MIDI sources. Each source is then distinguished by MIDI channel ranges. The channel offset is displayed in the rightmost column of the dialog box; you can change the channel offset in the Midi Setup... dialog.

Rules for Ports of MIDI Objects

The following table shows which port a MIDI object uses based on its typed-in argument in various situations.

object, argument	*OMS*	*MIDI Manager*	*Max MIDI driver*
no argument (**midiin**, **sysexin**, **rtin**, **midiout**)	First device listed in Midi Setup... dialog and object's Choose Port dialog	Port a	Modem port
no argument (all other MIDI objects)	All devices; ports are distinguished by MIDI channel offset	All ports a-f; port a is 1-16, b is 17-32, etc.	Modem port (1-16) and Printer port (17-32) if enabled
invalid name or abbreviation	First device listed in Midi Setup... dialog and object's Choose Port dialog	All channels on port a	All channels on the Modem port
channel number (**midiin**, **sysexin**, **rtin**, **midiout**)	All channels on the first device listed in Midi Setup... dialog	All channels on port a	All channels on the Modem port
channel number (all other MIDI objects)	The specified channel on a port calculated according to the Channel Offset in the Midi Setup... dialog	Port a if 1-16, b if 17-32, c if 33-48, d if 49-64, e if 65-80, f if 81-96	Modem port if 1-16, Printer port if 17-32
valid port name plus channel number (**midiin**, **sysexin**, **rtin**, **midiout**)	All channels on the specified port	All channels on the specified port	All channels on the specified port
valid port name plus channel number (all other MIDI objects)	The specified channel (mod 16) on the specified port	The specified channel (mod 16) on the specified port	The specified channel (mod 16) on port a (Modem) or port b (Printer)
invalid port name plus channel number (**midiin**, **sysexin**, **rtin**, **midiout**)	All channels on the first device listed in Midi Setup... dialog	All channels on port a	All channels on the Modem port
invalid port name plus channel number (all other MIDI objects)	The specified channel (mod 16) on the first device listed in the Midi Setup... dialog	The specified channel (mod 16) on port a	The specified channel (mod 16) on the Modem port

Embed a patcher inside a box

bpatcher

bpatcher is a user interface object which allows subpatchers that are both visible and operable inside a patcher window. You can group a collection of user interface objects in common configurations and re-use them for common needs. Or you can use the box as a the changing visible window on a larger collection of user interface tools. Look at the example below which shows how to use the offset message to the **thispatcher** object to change the visible part of the patcher inside the box.

Input

If the patcher being used in a **bpatcher** contains **inlet** objects, they will appear in left-to-right correspondence as inlets in the **bpatcher** object's box.

(mouse) When the window containing the **bpatcher** is locked (or the command key is held down) and the mouse is clicked inside the **bpatcher** object's box, the gesture is handled by the **patcher** inside the box.

If the shift and command keys are held down while clicking on a **bpatcher**, dragging the mouse moves the upper left-hand corner of the visible part of the **patcher** inside the box. The assistance area of the patcher window shows the pixel values of the offset.

If the command and option keys are held down while clicking in a **bpatcher**, a pop-up menu allows you to open the original file of the **patcher** contained inside the box in its own window, or change the **patcher** currently contained in the box. If no **patcher** is currently embedded, a standard open file dialog appears for loading a patcher file into the box.

Arguments

(Get Info...) When the **bpatcher** object is created or you select a **bpatcher** and choose Get Info... from the Max menu, a standard open file dialog is brought up prompting you for the name of a patcher file to display inside the box.

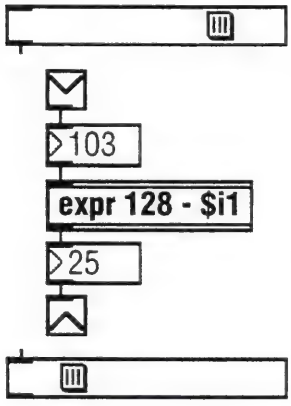
Output

If the **patcher** being used in a **bpatcher** contains **outlet** objects, they will appear in left-to-right correspondence as outlets in the **bpatcher** object's box.

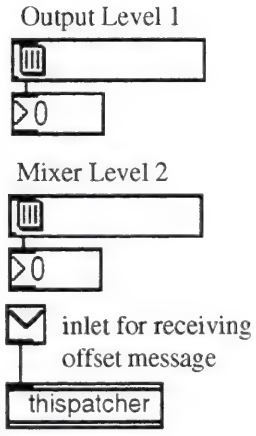
bpatcher

Embed a patcher inside a box

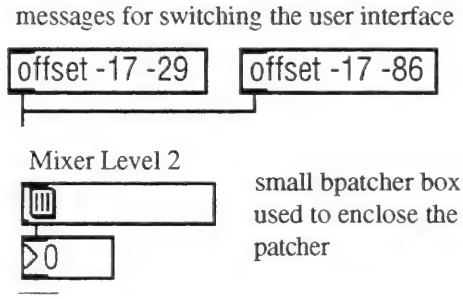
Examples



A patcher that inverts its input



Patcher for switching user interface with offset messages


messages for switching the user interface
offset -17 -29
offset -17 -86
Mixer Level 2
0
small bpatcher box used to enclose the patcher

Using the switchable patcher inside a bpatcher

Issues

User interface objects which are partially outside the **bpatcher** object's box will not redraw in response to messages received in their inlet.

See Also

patcher	Create a subpatch within a patch
Encapsulation	How much should a patch do?

Pass a number from outlet to outlet, out each one in turn

Bucket

The restriction on the number of outlets (specified by the typed-in argument) has been removed. **Bucket** also has several new operation modes.

Input

freeze	Suspends the **Bucket** output, but new numbers continue to shift values internally.
thaw	Resume **Bucket** output.
roll	Causes **Bucket** output in a loop.
L2R	Sets **Bucket** to output from left to right (the default).
R2L	Sets **Bucket** to output from right to left.

Before version 2.2, **coll** did not associate data with index 0. Now it will associate data with index 0, with one limitation: an index of 0 and a symbol cannot *both* be associated with data.

The maximum size of a list stored in a **coll** is now 250 items.

New Messages for Sorting and Rearranging Data in a Coll

sort — The sort message takes two arguments. If the first argument is -1, the items in the **coll** are sorted in ascending order. If the first argument is 1, the items in the **coll** are sorted in descending order.

The second argument specifies what is used to sort the contents of the **coll**. If the second argument is -1, the index (or symbol) associated with the data is used. If the second argument is not present or 0, the first item in the data is used. If the second argument is 1 or greater, the second (or greater) item in the data is used.

When sorting the data, symbols are alphabetized and numbers are arranged by value (how else would you sort numbers?).

swap — The swap message takes two symbols or two numbers, and exchanges the data associated with each symbol. For example, if the **coll** contains...

1, 400;
2, 700;

swap 1 2 would change the **coll** to...

1, 700;
2, 400;

subsym — The subsym message is used to change the symbol associated with data. The first argument is the new symbol to use, and the second argument is the symbol associator to replace. For instance, if the **coll** contains...

foo, 40 50 60;

subsym jack foo will change the **coll** to...

jack, 40 50 60;

renumber — Makes the numbers associated with the **coll**'s data consecutive and increasing. The argument to the renumber message specifies the starting number address for the data. Here's a before and after example for **coll** sent the message renumber 1.

Before	After
4, apple;	1, apple;
6, banana;	2, banana;
3, cherry;	3, cherry;
9, durian;	4, durian;

Send a stream of numbers to individual outlets

cycle

Input

int or list — The stream of numbers to be directed to successive outlets.

Arguments

int — Optional. Determines the number of outlets. The default is one outlet.

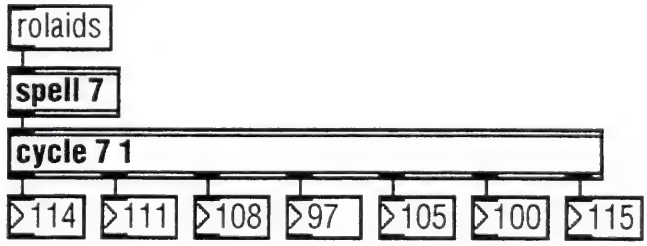
int — Optional. Sets the output mode. If non-zero, **cycle** detects separate "events" and restarts at the leftmost outlet when a new event occurs. Examples of separate evens include messages with delays in between them, and messages triggered by successive mouse clicks or MIDI events. A stream of items separated by commas in a message box would be considered a single event. If this argument is not present or zero, the values cycle through all the outlets, regardless of whether they are attached to separate events or not.

Output

int — Out any outlet: In *cycle* mode, each successive number received, either separately or as part of a list, is directed to an outlet to the right of the previous number. When the cycle reaches the rightmost outlet, the next number is sent out the left outlet.

In *event-sensitive* mode, any number which is a new event restarts the output at the left outlet.

Examples



Using cycle to get ASCII relief

See Also

spell — Convert input to ASCII codes
Bucket — Pass a number from outlet to outlet, out each one in turn

Script-configurable envelope editor

env

The **env** object is a script-configurable user interface for function editing, oriented toward the task of editing envelope data in synthesizer patch editors.

There are two flavors of this object—**env** displays and edits the envelope in its own windows, while **envi** (pronounced "envy") is a user interface object which allows an envelope to be seen inside a patcher window. Unless otherwise noted, both objects will be referred to generically in the documentation as the **env** object.

The **env** object is configured by a *script*—a text file—which defines the number of points in an envelope and associates them with some number of data values. If the script is read in successfully (i.e. it contains no syntax errors), the user should be able to change displayed data points in the **env** window. **env** saves the name of the last script file read and will try to locate it the next time its owning patcher is loaded.

Arguments

symbol — The **env** object takes an optional argument which is a symbol that names a script file to be read in which will define the behavior and appearance of the envelope.

The **envi** object, since it's a user interface object, doesn't have a typed-in argument. However, in both the **env** and **envi** objects, the name of the last script file read in is saved in the patcher file containing the object.

A new script file can be opened with the read message. And selecting the **envi** object and choosing **Get Info...** from the File menu puts up standard open file dialog for selecting a new script file to be read in.

Structure of an Envelope

The envelope is defined by a set of hierarchically arranged script messages. Both **env** and **envi** use identical format for script files.

Each **env** object consists of a window (technically in **envi**, a box in a patcher window), a number of groups, each of which contain points which are logically connected. Each point contains horizontal and/or vertical aspects, and each aspect can contain one or more display scales, which map internal data values to those displayed on the legend of the envelope window.

Script Messages

The format of a script file is similar to the script for a **lib** object. It consists of #E followed by a message keyword (such as group or point), followed by that message's arguments. See the Script Examples section below for examples.

The window message

Defines parameters applying to the entire **env** object and its display.

symbol — 1. Title of the envelope window (doesn't apply to **envi**). To use spaces in the title, use single "smart" quotes (option-right bracket and option-right brace).

int 2. Horizontal size. Size of the window (or box, in the case of **envi**) in pixels. For the window, the size will be actually be 15 pixels larger to accommodate the scroll bars.

int 3. Vertical size.

int 4. Number of groups. Each group will be defined in subsequent group messages (see below).

int 5. Number of data values that define the envelope(s).

int 6. Left margin. Distance in pixels from left edge of the window (box) where the envelope and text legend is drawn.

int 7. Bottom margin. Distance in pixels from bottom edge of the window (box) where the envelope is drawn.

int 8. Top margin. Distance from the top of the window (box) where the envelope is drawn. This should take into account the legend (which is 15 pixels), so a value of 20 or more pixels is suggested.

The group message

Defines a group of logically connected points, what would usually be thought of as an "envelope"—but the **env** object allows an arbitrary number of groups in a single window.

int 1. Group number. Specifies the group (starting at 1) being defined.

symbol 2. Group name. Precedes the name of any specific parameter and value in an envelope legend display. The word none can be used to indicate that no group name is desired.

int 3. Number of points in this group. Each will be defined below with a point message.

int 4. Visible. 1 if this group is initially visible, 0 if it isn't.

int 5. Display flags. 1 if you only want the parameter names and values of a point being dragged. 0 if you want all the parameter names and values displayed when a point in the group is being dragged. Other display flags may be defined later.

The point message

Defines the appearance of a "point" in an envelope.

int 1. Point number being defined. The first point in any group is number 1.

int 2. Button size (in pixels) of the round or square "button" centered at this point.

int 3. Button flags. The rightmost bit (i.e. 0 or 1) is 0 if the button is to be square and 1 if the button is to be round. Bit 1 (i.e. 0 or 2) is 1 if the button is solid, 0 if it is

transparent. Other bits may be defined in the future to define the appearance of the button.

int 4. Line-from point. If non-zero, specifies another point, which should always be numbered less than this point, which is to be connected to this point with a line. This connection is only a display property. Logical dependencies between points are specified in the horiz and vert messages below.

The horiz and vert messages

These messages define the two directional aspects of each point. Most of the "meat" of the envelope specification is contained in these messages. If you wish to keep one of the directions fixed, you need not define that direction for a particular point. The arguments to horiz and vert are identical, except where noted.

symbol 1. Parameter name. The name (e.g.. 'Rate 1') associated with moving the point in this direction. none can be used if there is no parameter name associated with this point.

int 2. Data index. The index into the array of data values (starting at 0) corresponding to the value of this parameter. If there is no data associated with this direction, use -1 (this will not be uncommon for one or more directions of one or points in an envelope). When a list containing this data index and a value is sent to the **env** object, this point will move accordingly.

Note that all data values are stored as integers. You can display a floating point number in the legend for this parameter by defining a scale expression or table (see the scale message below).

int 3. Minimum value of this parameter.

int 4. Maximum value of this parameter.

int 5. Initial value of this parameter.

int 6. Increment of this parameter. Not currently supported, should be set to 1.

symbol 7. Unit name. The units of this parameter (e.g.. ms for milliseconds or % for percentage). none may be used if the units are not tied to any particular units, such as the rate and level units on Yamaha synthesizers).

When two points are "tied together" in the horizontal or vertical direction it means that changes in one point are linked to others. Ties are expressed in terms of higher numbered points being tied to lower numbered ones. There are two types of ties—absolute and relative. An absolute tie means that a point changes its position on the screen to assume the exact value of another point. A relative tie, which is very common for horizontal aspects, means that the location of any point on the screen is based on a distance from another point. The common envelope shown in the second Script Example section below has point 2 with a relative horizontal tie to point 1, point 3 with a relative horizontal tie to point 2 (and hence to point 1), and point 4 with a relative horizontal tie to point 3. If point 1 is allowed to move left and right (as for example if there were an initial delay for the envelope, all the other points

would move as well. None of the points are vertically tied to each other, although in a DX7 envelope which has a non-zero final level, it is customary to tie points , points 1 and 4 would be absolutely vertically tied. You cannot tie the horizontal direction of one point to the vertical direction of another.

int 8. Absolute tie point. Point number that this point is absolutely tied to (must be less than this point number). This point will appear at the exact same horizontal or vertical position as the point it is tied to. Use 0 if this point is not tied.

int 9. Fixed. If this point is fixed at a particular position on the screen, use 1. Otherwise use 0. This may be true for the horizontal or vertical direction of the first (leftmost) point in an envelope.

int 10. Relative tie point. Point number that this point is relatively tied to (must be less than this point number) in this direction. This point's position will be an offset (depending on its value) from the position of the point being tied to in the horizontal or vertical direction. Use 0 if this point is not relatively tied to other points in this direction (commonly true for the vertical direction).

int 11. Positive direction. Sets which direction the value of a point increases. For the vertical direction, 0 indicates that the value increases as the cursor is moved to the top of the screen, while 1 indicates that the value increases as the mouse is moved to the bottom of the screen. For the horizontal direction, 0 indicates that the value increases as the cursor is moved to the left, while 1 indicates that the value increases as the cursor is moved to the right.

int 12. Coverage size. Determines how many pixels the range of the parameter is mapped into. For a garden variety envelope, you generally use most of the entire vertical space for the vertical direction, so you would use a formula like: <window vertical size> - <legend height> - <top margin> - <bottom margin>

For the horizontal direction, the amount of space you use should be determined by the number of points in the envelope, and how much scrolling you want to require the user to do if the envelope is stretched to its maximum width.

The scale message

Defines a conversion between the internal values (integers) used to store the data in an envelope and their displayed values, which may be floating point numbers. When envelope parameters represent physical quantities, manufacturers often use scale factors. In the scale message, you can specify a mathematical expression to convert the internal format to another integer or floating point number which is displayed in the legend.

A scale can be expression in the form of the arguments to the **expr** object, or it can be a list of values (including symbols) to which the internal data values map.

Each direction can have an arbitrary number of scales, each of which is applicable over a specified range. If there is no scale which applies to a data value, the legend will display the internal data value. One use of a scale in this context might be if the lowest value of an envelope signified "Off"—you could have a scale that mapped 0 to the word "Off" but left the other values unchanged.

int 1. Minimum. Lowest value for which this scale applies.

int 2. Maximum. Highest value for which this scale applies.

int 3. Floating-point digits. Number of digits after the decimal point used to display floating-point numbers in the legend.

symbol 4. The word is or table. Determines whether what follows is interpreted as a mathematical expression or a table of values used for mapping.

5. Additional data. For *expressions*: $i1 represents the internal data being mapped to the legend. Examples:

is $i1 * .07; Multiplies the internal value by a scale factor

is $i1 - 1; Subtracts 1 from the internal value

is ($i1-1)*.07; Compound expression

is 100 - $i1; Inverting an internal value

For *tables*: a list of values which map successive values of the internal data separated by spaces. The table can contain up to 240 elements. Use additional scale messages for larger tables. Example:

table Off 10 20 30 40;

Here, the minimum value will be mapped to the word "Off", next value to 10, next value to 20 etc.

Other Example scale messages:

#E scale 0 0 0 table Off; (Maps the minimum value to the word "Off.")

#E scale 1 10 2 is $i1 * .04; (Scales additional values by .04 and prints as floating-point number with 2 decimal places.)

The phase message

This message specifies that the previously defined vert aspect of a point has a signed component. Either the parameter of the envelope can be a negative number, or there is a separate data value that represents the phase (0 for negative, 1 for positive). The phase message must immediately follow the vert message it modifies.

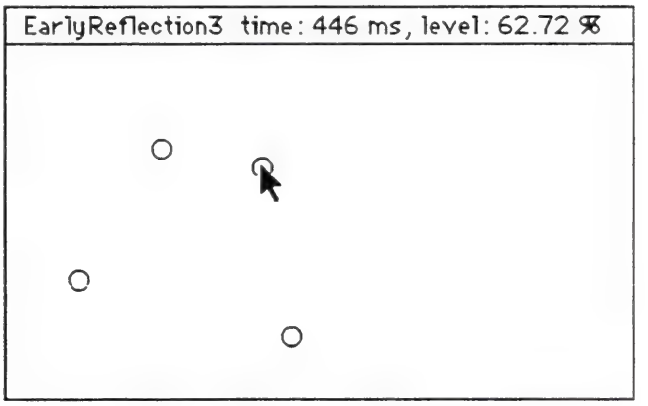
The end message

This message is required at the end of an envelope script. It reconfigures the **env** object and changes the display in its window or box if necessary. It has no arguments.

Script Examples

The following script defines an envelope which consists of 4 groups of individual points which are used in an early reflection tap editor. The horizontal position of the point determines a delay and the vertical position determines a percentage of the original signal to repeat. A picture is shown after the script.

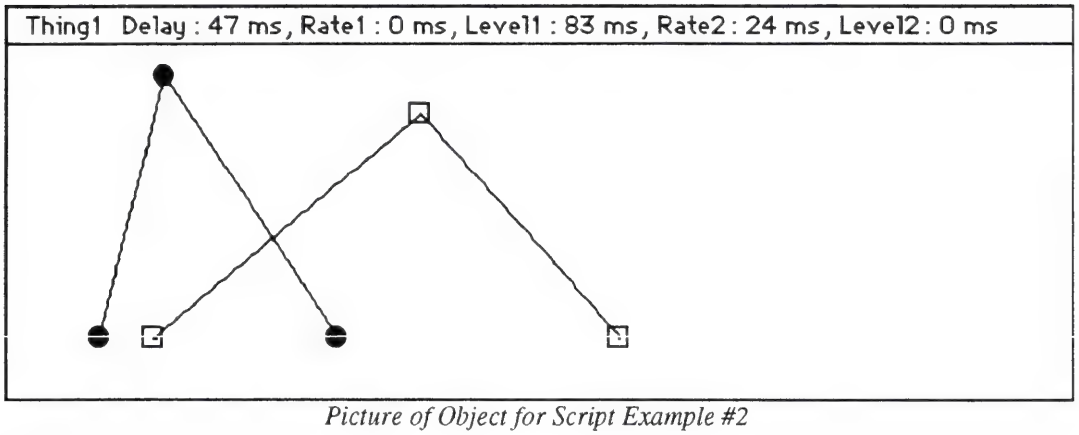
```
#E window ERFEnv 400 148 4 96 8 8 24;
#E group 1 EarlyReflection1 1 1 1;
#E point 1 8 1 0;
#E horiz time 0 1 500 1 1 ms 0 0 0 0 100;
#E vert level 1 0 1024 0 1 % 0 0 0 0 100;
#E scale 0 1024 2 is $i1 * .0977;

#E group 2 EarlyReflection2 1 1 1;
#E point 1 8 1 0;
#E horiz time 2 1 500 1 1 ms 0 0 0 0 100;
#E vert level 3 0 1024 0 1 % 0 0 0 0 100;
#E scale 0 1024 2 is $i1 * .0977;

#E group 3 EarlyReflection3 1 1 1;
#E point 1 8 1 0;
#E horiz time 4 1 500 1 1 ms 0 0 0 0 100;
#E vert level 5 0 1024 0 1 % 0 0 0 0 100;
#E scale 0 1024 2 is $i1 * .0977;

#E group 4 EarlyReflection4 1 1 1;
#E point 1 8 1 0;
#E horiz time 6 1 500 1 1 ms 0 0 0 0 100;
#E vert level 7 0 1024 0 1 % 0 0 0 0 100;
#E scale 0 1024 2 is $i1 * .0977;

#E end;
```



Picture of object for Script Example #1

The following script defines a two groups with more traditional synthesizer amplitude envelopes that have three points. The first point is fixed in the vertical direction but moves horizontally. The other two points move in both directions, and all three points are connected by a line. A picture is shown after the script.

```
#E window TestEnv 400 148 2 10 8 8 24;

#E group 1 Thing1 3 1 0;
#E point 1 8 0 0;
#E horiz Delay 0 0 99 0 1 ms 0 0 0 0 100;
#E vert none -1 0 99 0 0 none 1 0 0 0 100;
#E point 2 8 0 1;
#E horiz Rate1 1 0 99 50 1 ms 0 0 1 1 100;
#E vert Level1 2 0 99 50 1 ms 0 0 0 0 100;
#E point 3 8 0 2;
#E horiz Rate2 3 0 99 50 1 ms 0 0 2 1 100;
#E vert Level2 4 0 99 50 1 ms 0 0 0 0 100;

#E group 2 Thing2 3 1 0;
#E point 1 8 3 0;
#E horiz Delay 5 0 99 0 1 ms 0 0 0 0 100;
#E vert none -1 0 99 0 0 none 1 0 0 0 100;
#E point 2 8 3 1;
#E horiz Rate1 6 0 99 50 1 ms 0 0 1 1 100;
#E vert Level1 7 0 99 50 1 ms 0 0 0 0 100;
#E point 3 8 3 2;
#E horiz Rate2 8 0 99 50 1 ms 0 0 2 1 100;
#E vert Level2 9 0 99 50 1 ms 0 0 0 0 100;
#E end;
```



Picture of Object for Script Example #2

Input Messages

Because it can have an arbitrary number data values, the **env** object has only one inlet. The envelope data is stored in an array. The script file specifies how array indices correspond with horizontal and vertical aspects of the points in an envelope.

list A list received by **env** stores a new value in a data point. The first number in the list specifies the location (array index), and the second number is the data value to

store at the location. The **env** object limits the range of its input values, according to the minimum and maximum of each data point specified in the script file.

The **funnel** object takes a number in one of its inlets and outputs a list with the first element being the index of the inlet and the second element being the incoming number. It was designed to be used to prepare the lists required by the **env** object.

int — If the number is between 0 and the maximum array index, **env** outputs a list containing the index followed by the data value at the array index.

show — The word show, followed by an group number, makes that group visible. Followed by two numbers, makes a range of groups visible from the first to the second number.

hide — The word hide, followed by an group number, makes that group invisible. Followed by two numbers, makes a range of groups visible from the first to the second number.

open — Opens the **env** object's display window if its closed, or brings it to the front. Doesn't apply to the **envi** object.

read — Puts up a standard open file dialog for the user to select a new script file for configuring the object.

dump — Outputs all the current data values of the envelope, as successive two element lists. The first number is the data index and the second is the data value.

Output

list — When the mouse button is released or a number is received in its inlet, **env** sends lists outs its outlet which consist of two numbers. The first is an array index and the second is the new value at that index. Only newly modified values are output. When **env** receives the dump message in its inlet, all data values are sent out in this list format.

The **spray** object takes a list as input and sends the second element out the outlet number specified by the first element. It was designed to distribute the lists output by the **env** object to individual outlets for display by number boxes or to send to librarian editor objects such as **libto**.

Using an Envelope Window or Box

The envelope display has two areas separated by a horizontal line—the upper area of 15 pixels contains a legend of text in 9 point Geneva that indicates the names and values of the points the user is currently changing. The lower area contains the actual groups of points which may or may not be connected by lines.

The use of the **env** object's window (or the **envi** object's box) is simple—just click on one of the visible points. With no modifier keys held down, data values are incremented by a pixel's worth

of movement. How much this amounts to is determined by the ratio of each direction's Coverage size argument to its parameter range (difference between maximum and minimum values). For example, in the first example script above, there are 1024 data points and a Coverage size of 100, so moving the cursor one pixel changes the value by 1024/100, or about 10.

With the **shift** key down, movement of a point being dragged is constrained to the direction the cursor moves in first. Releasing the shift key at any time removes the constraint.

With the **command** key held down, mouse movement is in "fine mode"—no matter what the ratio of parameter range to Coverage size, the parameter data is changed by 1 with each pixel you move the mouse.

Fine mode can be entered or left instantaneously by pressing or releasing the command key while dragging the mouse.

See Also

funnel	Map a number to a list which identifies its inlet
spray	Distribute an integer to a numbered outlet

envi

Script-configurable envelope in a patcher window

The **envi** object is the patcher window version of the **env** object. The discussion of the **env** object covers both objects.

Display a picture from a PICT file

fpic

The **fpic** object includes pictures in patcher windows, as does the Paste Picture command in the Edit menu, but instead of including the picture in a patcher file, which is often slow to save and load, **fpic** just references a PICT file on disk. After placing an **fpic** object in a patcher window, choose Get Info... from the Max menu. A standard open file dialog appears; choose the desired PICT file. The picture will always appear at 100%, and the **fpic** object's box must always be resized manually. The lower right part of the picture is clipped by a box which is smaller than the size of the picture.

You can put a picture from the Clipboard into a Patcher by choosing Paste Picture... from the Edit menu. In this case, however, the picture is stored in the patcher, which may increase the time required to open a patcher file. Another advantage of using the **fpic** object is disk space and memory requirements—the same picture file may be referenced in many patcher windows, rather than being saved in each one.

Input

None.

Arguments

(Get Info...) Brings up a standard open file dialog box for choosing a PICT file to display inside the **fpic** object's box.

Output

None.

See Also

Menus	Explanation of commands

Map a number to a list which identifies its inlet

funnel

Input

int In any inlet: The number to be sent out in a list preceded by the inlet number.

Arguments

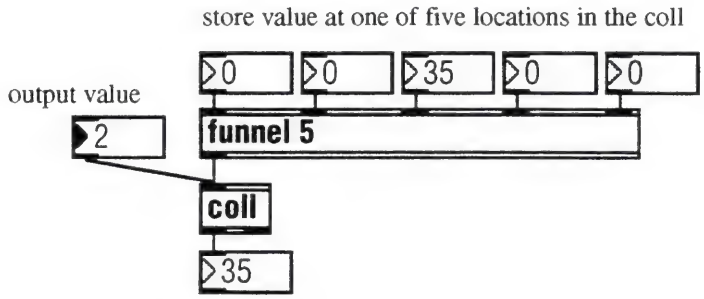
int Optional. The number of inlets in the **funnel**. Default is 2.

int Optional. The offset for the first inlet number. Default is 0.

Output

list When receiving a value in any inlet, **funnel** outputs a list consisting of the inlet number followed the value. **funnel** is designed for "funneling" many streams of numbers into the **env** or **envi** objects, but it can be useful in conjunction with other objects such as **coll** and **table**.

Examples



Using funnel to route data into a coll object

See Also

env Script-configurable envelope editor
envi Script-configurable envelope in a patcher window
spray Distribute an integer to a numbered outlet

Interface to
Mattel Power Glove

glove

The **glove** object works with a Mattel Power Glove connected to a Macintosh ADB port via a GoldBrick™ interface.

Input

bang — Polls the Power Glove and reports its current position out the outlets.

init — Sends an initialization command to the GoldBrick driver and interface.

Arguments

None.

Output

int — Out left outlet: X position of the Power Glove. From the perspective of someone facing the screen, 0 is to the right and 4095 is to the left.

Out 2nd outlet: Y position of the Power Glove. From the perspective of someone facing the screen, 0 is the lowest position, 4095 is the highest.

Out 3rd outlet: Z position of glove. 4095 is nearest the screen, 0 is farthest away.

Out 4th outlet: Rotation, 8 different values in increments of 120.

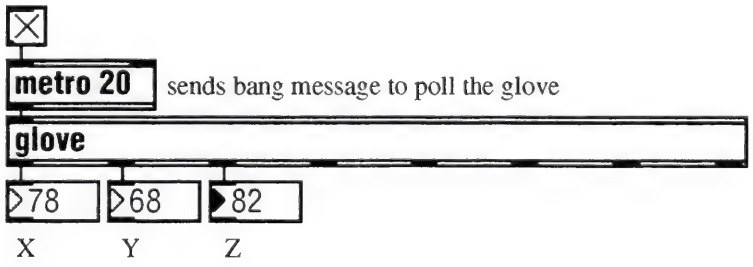
Out 5th outlet: Bend of thumb. 0 is no bend, 3 is maximum bend.

Out 6th outlet: Bend of index finger. 0 is no bend, 3 is maximum bend.

Out 7th outlet: Bend of middle finger. 0 is no bend, 3 is maximum bend.

Out 8th outlet: Bend of ring finger. 0 is no bend, 3 is maximum bend.

Examples



Polling the Power Glove and reporting positional information

Play a Quicktime movie in a patcher window

imovie

The **imovie** object is the patcher window version of the **movie** object. The discussion of the **movie** object covers both objects. The **imovie** object accepts one additional message.

Input

border — border 0 sets the object not to draw a black line border around its movie. border 1 draws the border (this is the default).

See Also

movie — Play a Quicktime movie in a window
playbar — Quicktime movie play controller

Input

get — The get message to **lib** takes two arguments. The first specifies a patch number (starting at the **lib**'s first patch number) and the second an offset into the patch. **lib** sends a message to a **receive** object named in an argument to the getwho message the value of the patch at that location.

getwho — The getwho message specifies the name of the **receive** object that will receive the result of subsequent get messages sent to the **lib** object. For example, sending getwho bob followed by get 4 20 sends byte 20 of patch 4 to all **receive bob** objects.

set — The set message to **lib** assigns values to pseudo-variables. For example, set 5 2 would set the $i5 variable (the current MIDI channel) to a value of 2. The $i6 variable (current patch number) is now fully implemented.

names — The names message dumps a list of the current patch names in the format accepted by the **menu** object out the rightmost outlet.

sendselected — If there is a selected patch, this message forces it to be sent out the outlet.

Output

bang — Out middle outlet: **lib** sends a bang when an operation such as loading or sending a bank of patches is completed.

clear, append — The right outlet can set a **menu** object with the list of patch names when **lib** receives the names message.

Send a bang after all librarian objects have initialized

libbang

Input

None.

Arguments

None.

Output

bang — The **libbang** object outputs a bang message after all **libfrom**, **libswitch**, and **libtofrom** objects have sent out their values representing the contents of librarian's patch data. Each **libbang** sends out a bang after all the objects in its window are initialized, but if a patcher contains a subpatcher with editor objects, the **libbang** in the outer patcher will output before the editor objects in the subpatcher are initialized, and also before **libbang** objects in the subpatcher output a bang.

See Also

libfrom	Output stored patch value when editor is loaded
libtofrom	Store and retrieve patch values
libswitch	Change the patch being edited within an editor window
libnames	Output a librarian's patch names for a menu

Output a librarian's patch names for a menu

libnames

Input

bang — Sends out messages to assign a **menu** object with a list of the patch names in the current **lib** bank. It works identically to the names message to the **lib** object, but it is intended for creating a pop-up menu of patch names inside an editor.

int — 1 is equivalent to the bang message. 0 only sends output if the names have changed since the last time the **libnames** object sent the patch names out its outlet.

Arguments

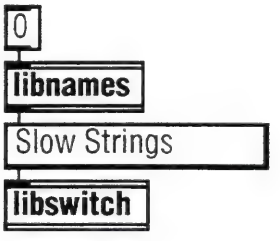
None.

Output

clear — The first message sent out by **libnames** is clear, to erase the previous contents of a **menu** object.

append — Each name is added to the **menu** object with an append message with the patch name as an argument.

Examples



*Using **libnames** to make an interface for selecting new patches to edit from within an editor.*

See Also

Librarians	Creating Patch Editors and Librarians
lib	Script-driven patch librarian
libswitch	Change the patch being edited within an editor window

Change the patch being edited within an editor window

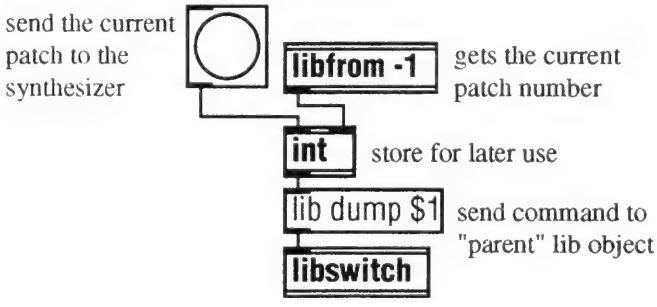
libswitch

Input

link The link message takes the name of a patcher file as an argument. If the patcher file can be found, it is loaded and all of the patcher's **lib**-related objects are linked into the editor "context" which contains the libswitch object. The link message to **libswitch** can be used to open a component of an editor only needed at a specific time, reducing the initial load time of an editor. You can use the capabilities of the **thispatcher** object (see below) to dispose of the editor once it has been loaded, or, if the user closes the main window of the editor patcher, any linked patchers will be disposed of automatically.

lib The lib message sends its arguments to the "parent" **lib** object of a **libswitch**. For example, you could force a patch to be sent to the synthesizer with the example shown below.

Examples



Store and retrieve patch values

libtofrom

libtofrom combines the **libto** and **libfrom** objects into a single object. Unlike those two objects, it can also change its arguments with the set message, making it useful in writing editors for devices that contain identical "sub-modules."

Input

int — The inlets of **libtofrom** are the **libto** part, receiving numbers and setting bytes in the librarian patch buffer and optionally sending out parameter change messages.

Two messages unique to **libtofrom** make it more powerful for patch editors that need to change dynamically.

set — The word set, followed by the standard **libto** or **libfrom** argument specification, replaces the initial typed-in arguments with a new specification. This message is useful if you are writing an editor which contains a set of identical "sub-modules" that are distinguished by their memory locations within the patch buffer and/or their parameter numbers. You can write one editor user interface for all such modules, and use set messages to change the **libtofrom** objects to peek and poke different areas of a patch's memory.

bang — The bang message outputs the current value at the specified patch offset(s). This is handy to use immediately after a set message to report the newly specified patch value to the user interface.

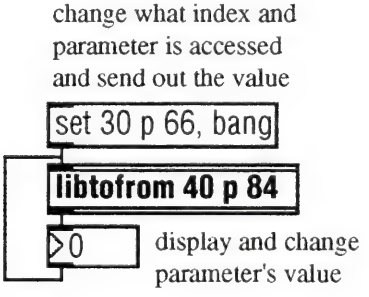
Arguments

The **libtofrom** object's arguments are identical to those of **libto** and **libfrom**.

Output

int — The outlets of **libtofrom** are the **libfrom** part of the object, sending out the current value of the patch buffer at the specified byte offsets.

Examples



See Also

libto — Store data into an edited patch, send parameter changes
libfrom — Output stored patch value when editor is loaded

Pop up menu, to display and send commands

menu

The **menu** object features two new interaction modes: scrolling and label. Scrolling mode makes the **menu** object work like a Numerical scrolling through the items in the menu. Label simply displays the text of the menu without drawing a box around it, and ignores any user clicking.

The storage scheme of the **menu** object has been completely rewritten for version 2.5, so that the setitem, append, clear, and delete messages immediately change the contents of the menu, even at interrupt level. Pop-up menus also draw much more quickly. The tradeoff for this new scheme is the requirement that you specify a maximum number of items in the Get Info... dialog. This number is adjusted automatically when reading in an old file which did not specify a maximum number of items. The default (and minimum) number of items is 32.

Input

delete — Followed by a number of an item, deletes that item from the **menu**.

set — **menu** now accepts a symbol argument to the set message. It searches for a menu item that begins with the symbol. If one is found, the **menu** is set to display that item.

symbol — The symbol message is identical to the set message with a symbol argument, except that the item number is output, along the text of the item as a message if the Evaluate Item Text feature is enabled.

Put up a custom menu bar

menubar

The **menubar** object provides control over the Macintosh menu bar. It allows your patcher to put up its own menus, and add items to standard File and Edit menus. When a menu item is chosen, the item number is sent out the outlet corresponding to the menu containing the item. You configure the **menubar** by writing a script in a text editor window available by double-clicking on the object.

Input

int — 1 displays the **menubar** object's menus, 0 restores the previous contents of the menu bar (either the Max menus or the menus of another **menubar** object).

checkitem — Followed by a menu number, an item number, and a code 0 or 1, checkitem puts a check before the specified item if the code is 1, otherwise it removes the check.

enableitem — Followed by a menu number, an item number, and a code 0 or 1, enableitem enables the specified item if the code is 1, otherwise it disables the item.

markitem — Followed by a menu number, an item number, and an ASCII character code, markitem places the character next to the specified item. Common mark character ASCII codes are 18 for the check mark and 19 for the diamond mark. You may also wish to use the em dash (209) or filled-in dot (165).

Arguments

int — Optional. Sets the number of menus in the object's menu bar. If present, it must be at least 5 (one additional menu). The four default menus are Apple, File, Edit, and Windows.

Script Messages

You define a **menubar** with a series of script messages, typed into a text editor window opened by double-clicking on a **menubar** object. When the window is closed and you confirm saving the changes, the script file is interpreted. If there are no errors, the menu will be ready for use when the **menubar** receives a 1 in its inlet.

Each message should be preceded by #X and end with a semicolon. The first script message must be apple and the last end. An example script follows the definition of the messages.

Messages to Modify Standard Menus

Message — *Arguments*

apple — Text of the first Apple menu item (i.e. About My Program...).

file — • Item number to output
• Text of item to add to file menu

The file message inserts items into the standard file menu after the Save As... item and before the MAX menus item. Any additional items are surrounded by blank

lines. Each item has a number associated with it which is sent out the File menu outlet when the item is chosen.

edit
- Item number to output
- Text of item to add to edit menu

The edit message inserts items into the standard Edit menu after the Clear item and before the Overdrive and Resume items. Any additional items are surrounded by blank lines. Each item has a number associated with it which is sent out the Edit menu outlet when the item is chosen.

saveas
- Item number to output.

The saveas message directs Max to send the specified number out **menubar**'s File menu outlet when the user chooses Save or Save As... from the File menu, instead of performing the standard Save actions. The number sent out the outlet when Save is chosen will be 1 less than the number sent when Save As... is chosen.

Messages for Creating New Menus and Items

Message *Arguments*

menutitle
- Menu number (must be at least 5 and <= argument to menubar)
- Name of menu

The menutitle message adds a new menu after the Windows menu in the menu bar. Notice that the last sentence uses the word 'menu' four times. The first additional menu is number 5. The menu number determines both the order of the additional menu in the menu bar and the outlet it uses when the user chooses its items. A menutitle message must precede any item messages that refer to its menu number.

item
- Menu number
- Item number output
- Text of item

The item message added an item to an additional menu previously defined with a menutitle message. Items are listed in the order they appear in the menu. The item number argument only specifies the number which is sent out the **menubar**'s outlet when the user chooses this item. It's a good idea to start your item numbers at 1 and list the items in the order you want them to appear in a menu. Note that the item text will put slashes and other escape characters recognized by AppendMenu (see Inside Macintosh, Volume I, pp. I-346 - I-349) into the menu item itself. Thus, the text "Washer/Dryer" will appear as "Washer/Dryer" not as "Washerryer ⌘D."

Completing the Script Definition

Message	*Arguments*
end	(none)

The end message builds the menus and reports any errors encountered.

Output

int — The default **menubar** object has four outlets. When activated (by receiving a 1 in its inlet), the leftmost outlet sends a 1 if the first item in the Apple menu is chosen. The second outlet sends the item number if an extra item is chosen from the File menu. The third outlet sends the item number if an extra item is added to the Edit menu. The fourth outlet sends an item number if the user chooses an item from the Windows menu. If additional menus have been defined, item numbers are sent out the additional outlets to the right, starting with the fifth one.

Examples

Here is an example **menubar** script:

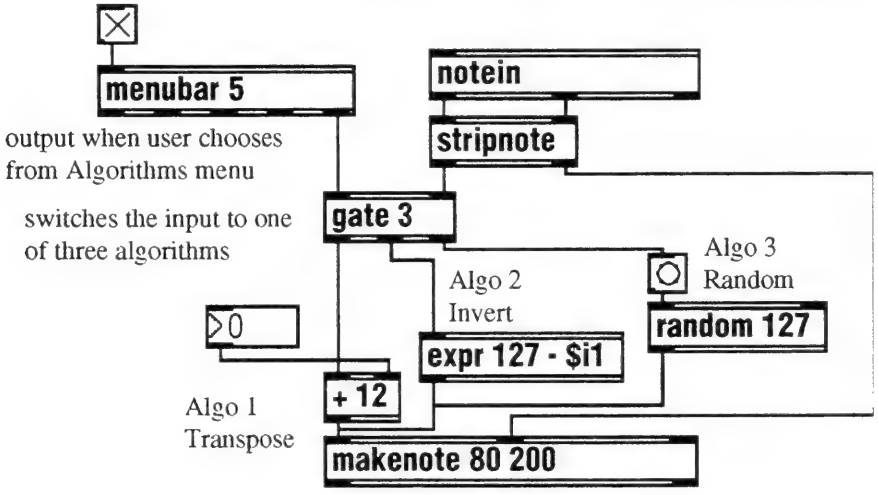
```
#X apple About Note Algorithms...;
#X saveas 5;
#X menutitle 5 Algorithm;
#X item 5 1 Transpose;
#X item 5 2 Invert;
#X item 5 3 Randomize;
#X end;
```

Note that we suggest capitalizing each letter in a menu item to maintain a consistent style with other items in the menu.

The above script is used in a **menubar** in the following example, which uses the extra menu to switch among three note-processing algorithms.

menubar

Put up a custom menu bar

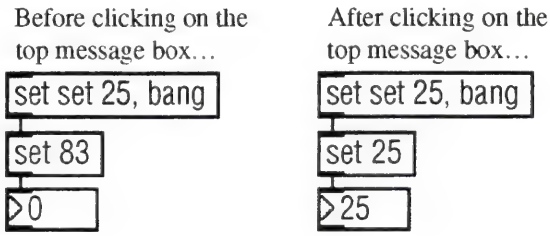


See Also

menu	Pop-up menu, to display and send commands
Menus	Explanation of commands

Real-time Message Building

One of the most common complaints about Max has been addressed in version 2.5. This was that the set, append, and prepend messages to the **message** box did not change the text in real time, so that one could not immediately send the contents of a **message** box after changing the text. This is no longer the case: the message box text changes immediately. The example shown below will work "as intended" in version 2.5.



Input

open The open message brings up a dialog box allowing you to change the contents of any **message** box, or choose Get Info... from the Max menu.

The first (optional) argument to the open message is a symbol which is the prompt for the dialog box. The default prompt is "Message Text." Use single smart quotes to use spaces in the message. If the symbol is followed by a 1, the contents of the **message** box will be send out its outlet after it is edited and the user clicks OK. By default, the contents of the **message** box are not sent out its outlet.

msd can send and receive samples using the SCSI-based SMDI protocol. The new messages for this type of sample transfer were included in version 2.2 but in error, the help file which described them was not included. The messages are described below and the new help file is included in version 2.5.

Input

scsiid — Followed by a number between 0 and 6, sets the SCSI ID of the SMDI-compatible sampler. Subsequent scsisend and scsiload commands (see below) will be directed at a device set to that SCSI ID.

scsisend — If a sample has been loaded or read into the **msd** object, scsisend, followed by a number, sends the sample to the SCSI sampler at the sample location specified by its argument.

scsiload — Followed by a number, scsiload transfers a sample at the location in the SCSI sampler specified by the number into the **msd** object, obliterating any previously stored sample. If the transfer is successful, the sample can be written to a file with the write message.

Output

bang — Out middle-right outlet: sent when an operation is successfully completed.

Out right outlet: sent when an operation failed.

Play a Quicktime movie in a window

movie

The **movie** and **imovie** objects require the Quicktime init. The object will not be created if Quicktime is not installed. **movie** plays a Quicktime movie in its own window. **imovie** plays a Quicktime movie in a box inside a patcher window.

Input

int — Sets the current time location of the movie. If the movie is playing, it will play from the newly set location. 0 is always the beginning. The end time varies from one movie to another. The length message reports the end time location out the left outlet.

active — Followed by a 1, active makes the movie active (the default). Followed by a 0, active makes the movie inactive. An inactive movie will not play or change location.

mute — Followed by a 1, mute turns off the movie's sound (if it has any). Followed by a 0, mute turns on the movie's sound (the default).

next — Followed by a number, next moves the time location ahead by that amount. If no number is supplied, next moves the time ahead by 5.

prev — Followed by a number, prev moves the time location backward by that amount. If no number is supplied, prev moves the time backward by 5.

rate — Sets the playing speed of the movie. If rate is followed by one integer, that number is taken to be a whole number playing speed. If followed by two numbers, the first number is taken to be the numerator and the second the denominator of a fractional speed. 1 is the normal playing speed, 0 means the movie is stopped, and a negative rate plays backwards. rate 1 2 would play the movie at half speed. Immediately after you send a non-zero rate message, the movie will begin playing, so you may wish to precede any rate messages with an integer to locate to the desired starting position.

read — Followed by a symbol, read looks for a Quicktime movie file with the specified name in the Max search path and opens it if the file exists, closing any previously open movie file. With no arguments, read puts up a standard open file dialog box and reads in any movie file you select.

reload — Reloads the current movie file, which may free up memory. This message seems to be unnecessary for the release version of Quicktime.

start — Sets the movie's rate to 1 and begins playing from the beginning.

time — Reports the current time location of the movie.

length — Reports the end time location of the movie.

vol — Followed by a number, vol sets the movie's sound volume. A negative number mutes the sound, while the maximum volume is 255.

movie

Play a Quickime movie in a window

Output

int Out left outlet: The current time location when receiving the time message.

Out middle outlet: The horizontal position of the mouse, relative to the left side of the movie box or window, when the mouse is clicked or dragged inside the movie.

Out right outlet: The vertical position of the mouse, relative to the top of the movie box or window, when the mouse is clicked or dragged inside the movie.

See Also

imovie Play a Quicktime movie in a patcher window

Set pop-up menu with names of MIDI devices

omsinfo

Before version 2.2, **omsinfo** required OMS; now **omsinfo** works whether you are using OMS or not, providing a tool to create a pop-up menu interface for choosing input and output ports which will work with any MIDI environment you choose to use with Max. We strongly recommend that **omsinfo** be used in any Max MIDI applications which will be widely distributed. Patchers which assume the old modem-port-only environment are particularly infuriating to use with OMS. Please look at the sample applications for examples of using **omsinfo** in user interfaces.

Input

controllers — **omsinfo** takes the controllers message in its left inlet for producing a list of OMS input devices marked by the user in the OMS Setup application as *controllers*. The controllers message will produce a subset of the list of all input devices produced by sending a message in **omsinfo**'s right inlet. The controllers message can be useful when choosing a port to use with **notein** or **ctlin**. Other input devices may be rack-mounted sound modules whose "input" to the computer is only used for system-exclusive dumps, so there's no need to list them as possible sources of control.

Combine numbers or symbols into a list or message

pack

pack can take an arbitrary number of arguments, and accepts symbols and lists in any of its inlets.

Each argument to **pack** creates an inlet, and the arguments become the initial values. Inlets can accept individual symbols, lists of numbers, or messages (symbols followed by other symbols or numbers). A bang or other message in the leftmost inlet produces a list (or message) at the outlet. If you send a list or multi-element message into any inlet, the first element is assigned to the inlet receiving the message, and subsequent elements are distributed to the inlets to the right of this one. If a **pack** object receiving the list or message runs out of inlets, the rest of the input is ignored.

Input

set **pack** accepts the set message (in any inlet, although the message's distinctive behavior is in **pack**'s left inlet). It allows assignment of symbols or numbers to **pack**'s inlets without producing any output.

pack is thus a more general list and message construction object; see the **pack** help file for examples.

pcontrol

Input

enable **pcontrol** can enable and disable the MIDI input and output to a patcher window. enable 1 enables MIDI, enable 0 disables MIDI. An additional argument determines whether the subpatchers of the patcher connected to **pcontrol** are enabled or disabled. If this second argument is 0 or not present, they are left alone. enable 1 1 will enable the connected patcher and all of its subpatchers, enable 0 1 will disable the connected patcher and all of its subpatchers.

load The word load, followed by a symbol which names the file, searches for a file in the Max search path with that name and loads it into memory.

Quicktime movie play controller

playbar

The **playbar** object implements the standard Quicktime movie controller for use with the **movie** or **imovie** objects. To use **playbar** with either of these objects, connect the left outlet of **playbar** to the inlet of **movie** or **imovie**. Then send the **playbar** object a bang; this will cause the **playbar** object to link up with the **movie** or **imovie** object. Linking will happen automatically when a patcher file containing connected **playbar** and **movie** objects is loaded. Thus, sending the bang to **playbar** is only necessary when you're building a patcher.

Currently, **playbar** only supports being connected to one **movie** object at a time. The connection must be direct—it cannot take place through a **send-receive** pair.

After **playbar** and **movie** are linked, any messages sent to the **movie** object which change its location or playing status are reflected in the **playbar** object. And of course, clicking on the gadgets in the **playbar** object will control the **movie** playback.

Input

bang — If a **playbar** object is connected to a **movie** or **imovie** object, bang connects the two objects together so the **playbar** can control the Quicktime movie, and changes to the movie will update the **playbar**'s gadgets.

Arguments

None.

Output

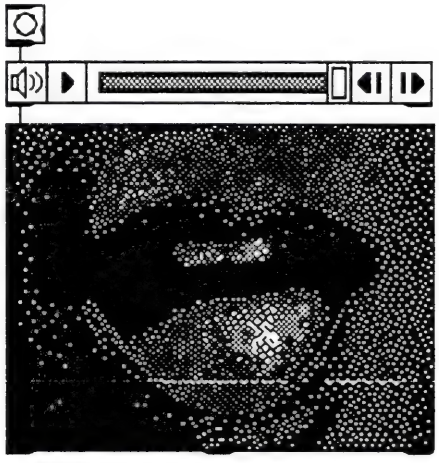
(internal) — Out left outlet: connect the left outlet of the **playbar** object to a **movie** or **imovie** object to establish a "connection" between **playbar** and a Quicktime movie.

int — Out right outlet: Each command processed by **playbar** is sent by number out its right outlet. A directory of command numbers and their meaning can be found in the Quicktime Standard Movie Play Controller documentation. By properly interpreting these commands, you might be able to use **playbar** for other purposes besides movie control. However, the "thumb" in the controller has no range until an associated Quicktime movie with a non-zero duration is linked to the play controller.

playbar

Quicktime movie play controller

Examples



Using **playbar** *with* **imovie**

See Also

movie	Play a Quicktime movie in a window
imovie	Play a Quicktime movie in a patcher window

Share variables specific to a patcher and its subpatchers

pv

pv operates identically to the **value** object, with two exceptions. First, **pv** objects that share the same name only share the same value if they are in the same patcher, or one of its subpatchers. Second, the **pv** object cannot be the receiver of a message sent in a **message** box (the first symbol after a semicolon). It all adds up to one thing: **pv** means *private value*.

Input

any message — The number or message is stored, to be shared by all other **pv** objects with the same name that are inside the object's patcher or its subpatchers (or, if in a subpatcher, its parent patchers). A message received in any other such **pv** object will change the stored message.

bang — Sends out the stored message, number or list.

Arguments

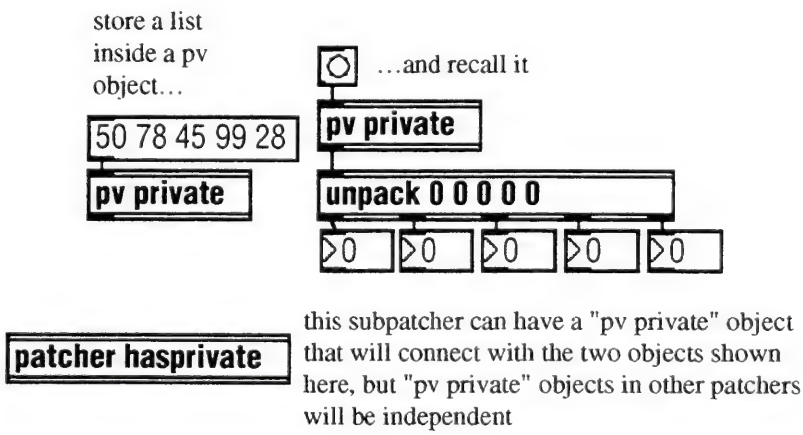
any symbol — Obligatory. Provides an identifying name.

any message — Optional. Initializes the stored contents of the **pv** object. Note that when two or more **pv** objects in a patcher file that share the same name are initialized to different values, the one which is initialized last determines the value. Since the order in which **pv** objects are initialized cannot be precisely determined, it is suggested that the best use for this feature is to initialize only one **pv** object.

Output

any message — A bang causes the stored message to be sent out.

Examples


store a list inside a pv object...
...and recall it
50 78 45 99 28
pv private
pv private
unpack 0 0 0 0 0
0
0
0
0
0
patcher hasprivate
this subpatcher can have a "pv private" object that will connect with the two objects shown here, but "pv private" objects in other patchers will be independent

See Also

value — Share a stored number with other objects
send — Send messages without patch cords
receive — Receive messages without patch cords

Play a sound through the internal speaker

snd

The **snd** object has been substantially improved. **snd** can play multiple sounds at once on certain machines. In addition, you have pitch transposition of samples and control over velocity and duration. These features require System 6.0.7 or later.

Multiple Sounds

On suitably fast Macintoshes with Apple Sound Chips (except the Macintosh LC and LC II), the **snd** object can play multiple sounds at once. On a IIci for example, four sounds can be played at once. On a Powerbook 140, 2 sounds can be played at once, and on a IIfx, six sounds can be played at once. In addition, the **snd** object can play in Overdrive mode.

Input

int — In left inlet: **snd** now accepts MIDI note numbers and transposes the current sample according to the pitch of the MIDI note. To maintain compatibility, 1 plays the sound at its default pitch and 0 turns the sound off.

int — In middle inlet: A number between 0 and 255 sets the volume of the sample being played.

int — In right inlet: Sounds can be played for a specific duration. The right inlet accepts an integer specifying the duration of the sound in milliseconds. If no duration is supplied, the sound will play until it is over.

info — The info message received in the left inlet causes the **snd** object to output commands to set a pop-up **menu** object with the names of the samples residing in the current file. After having enabled Evaluate Item Text in the **menu** object's Get Info... dialog, you can send the right outlet of this menu to a **snd** object to change the currently playing sample.

Convert input to ASCII codes

spell

Input

any symbol — The ASCII value of each letter, digit, or other character in the symbol is sent out the outlet, one character at a time.

int, float — The ASCII value of the number in decimal is sent out the outlet, one character at a time.

list — Each number is converted to ASCII as described above, and a space character (32) is placed between number strings.

any message — Elements of the message are converted to ASCII one character at a time, and a space character (32) is placed between them.

Arguments

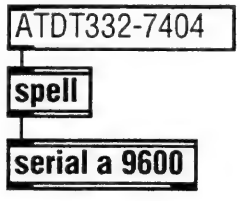
int — Optional. Sets the minimum output size. Any input that doesn't "spell" to the minimum length is followed by enough fill characters (the default is space, 32 in ASCII) to satisfy the minimum requirement.

int — Optional. If two arguments are present, the second specifies the fill character to use instead of 32. If you want to use 0 as a fill character, use any negative number as a second argument to **spell**.

Outputs

int — The ASCII representation of the input, one character at a time.

Examples



Using the **spell** *object, symbols, such as this modem command string, can be sent out the serial port in their ASCII representation.*

Distribute an integer to a numbered outlet

spray

Input

list — The first number in the list specifies the outlet number; the second is the data to send. If there are additional numbers in the list, they are sent out the outlets immediately to the right of the one specified by the first number in the list.

Arguments

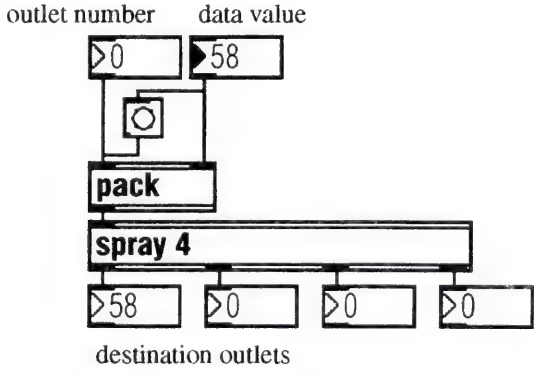
int — Optional. Sets the number of outlets. The default is 2.

int — Optional. Sets the offset for the first outlet number. The default is 0.

Output

int — When a list is received by the **spray** object, the first number specifies the outlet number, and the additional numbers are sent out that outlet. **spray** is designed for "spraying" numbers to separate outlets. In particular, you can connect the outlet of an **env** or **envi** object to the inlet of a **spray** object to distribute the envelope's values to separate outlets.

Examples



See Also

env — Script-configurable envelope editor
envi — Script-configurable envelope in a patcher window
funnel — Map a number to a list which identifies its inlet
route — Selectively pass the input out a specific outlet

Send messages to a patcher

thispatcher

The **thispatcher** object is placed *inside* the patcher you want to control.

Input

clean — Resets the patcher window's dirty flag, so the user won't be asked if she wants to save changes when the window is closed.

dispose — Permanently closes the patcher window and frees its memory. Use this in conjunction with the load message to the **pcontrol** object to open and close patchers under program control.

front — Brings the patcher window to the front, or opens the window and brings it to the front if it's not visible.

loadbang — Sending the loadbang message to **thispatcher** causes any **loadbang** objects to send out a bang. Any other objects which use loadbang message for initialization (such as the **preset** object) will receive this message as well.

offset — For patchers contained inside boxes (using the **bpatcher** object), the offset message sets the upper left corner of the visible portion of the patcher in the box. offset should be followed by two numbers; the first specifies the left offset (in pixels) and the second specifies the top offset. By default, patchers in boxes are displayed with an offset of 0,0. When you hold down the command and shift keys and drag in a **bpatcher** object's box, the offset changes as you move, and the current offset is displayed in the assistance area of the window which contains the **bpatcher**. You can use these numbers as arguments to the offset message.

write — Updates the patcher's file if it has a name; otherwise, brings up a Save As... dialog.

(others) — **thispatcher** will respond to messages to create new objects. The format of these message is obscure and subject to change, but you can get some idea of what might be worth trying by examining a patcher file as text, and trying any of the messages that begin with #P. Leave the #P out of the message you send to **thispatcher**. Use of **thispatcher** to create new objects is not supported.

Arguments

None.

Output

None.

See Also

bpatcher — Embed a patcher inside a box
patcher — Create a subpatch within a patch
pcontrol — Control a subpatcher

Generate random numbers without duplicates

urn

Input

bang — In left inlet. Triggers the random number.

clear — In left inlet. Clears the list of already generated numbers.

int — In right inlet. Sets the maximum for the random number generator. Numbers will be between 0 and the maximum minus 1 inclusive.

Arguments

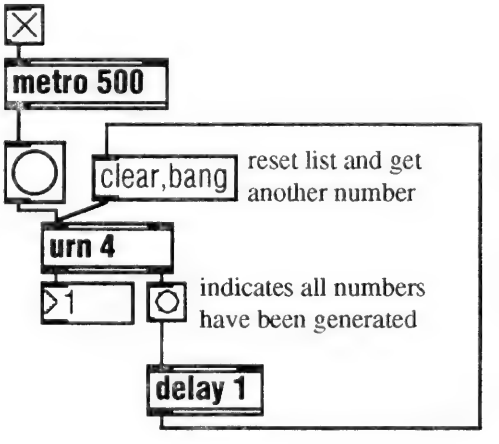
int — Obligatory. Maximum for random number generation.

Output

int — Out left outlet: If there are numbers within the current range that have not been sent out since the last clear message was received, **urn** generates a random number between 0 and the maximum minus 1.

bang — Out right outlet: When all numbers in the current range have been generated, **urn** sends a bang out the right outlet instead of a number out the left outlet.

Examples



Looping from 1 to 4 in a random order

See Also

drunk — Output random numbers in a moving range
random — Generate a random number

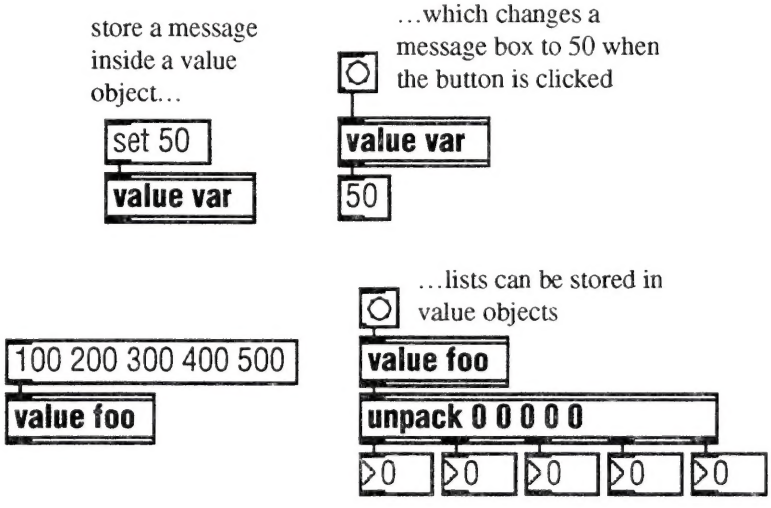
Input

The **value** object now accepts and stores any message, including lists.

Arguments

value now takes optional additional arguments after its naming symbol which initialize the value.

Examples



ARMA

Input

cmd — The cmd message can be used to send "primitive" commands consisting of ASCII codes to the video disk player. Commands usually consist of two-letter codes preceded by numeric arguments. For example, searching to frame 5000 could be accomplished with the message cmd 5000 SE. Refer to the owner's manual for your player for details. The cmd message is particularly useful with the Pioneer 8000 player, since it has a number of special features not supported by the regular messages of the **vdp** object.

setskip — Followed by a number, sets the number of frames to jump (forward or backward) from the current frame location when using the skip message.

skip — Followed by -1, skips backward by a number of frames specified in the setskip message. Followed by 1, skips forward by a number of frames specified in the setskip message.